

# Introduction to GAUSS

University of Pittsburgh, ECON 2150, Spring 2006  
compiled by Martin Burda

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Running GAUSS in Windows . . . . .	3
1.2	The Help Menu . . . . .	5
1.3	Syntax . . . . .	5
1.4	Note on Numerical Precision . . . . .	5
<b>2</b>	<b>Basics Matrix Operations</b>	<b>6</b>
2.1	Variables . . . . .	6
2.2	Creating matrices . . . . .	6
2.3	Referencing Matrices . . . . .	8
2.4	Reshaping Matrices . . . . .	9
2.5	Concatenation . . . . .	10
2.6	Matrix Operators . . . . .	10
2.7	Conformability and the "dot" operators . . . . .	11
2.8	Special matrix operations . . . . .	11
<b>3</b>	<b>Input and output</b>	<b>15</b>
3.1	Loading Gauss Data with <code>loadd</code> . . . . .	15
3.2	Using the Matrix Editor . . . . .	15
3.3	Text files ( <code>.txt</code> ) . . . . .	15
3.4	Spreadsheets . . . . .	17
3.5	Storing matrices ( <code>.fmt</code> files) . . . . .	18
3.6	Datasets ( <code>.dat</code> files) In Depth . . . . .	18
<b>4</b>	<b>Managing data</b>	<b>22</b>
4.1	<code>SHOW</code> , <code>PRINT</code> , <code>FORMAT</code> , <code>NEW</code> , <code>CLEAR</code> , <code>DELETE</code> . . . . .	22
<b>5</b>	<b>Statistical Functions</b>	<b>24</b>
5.1	Example 1 . . . . .	24
5.2	Example 2 . . . . .	24

<b>6</b>	<b>Flow of Control</b>	<b>25</b>
6.1	Conditional branching: IF . . . . .	25
6.2	Loop statements: DO WHILE/UNTIL and FOR . . . . .	26
<b>7</b>	<b>Suspending execution</b>	<b>29</b>
7.1	Temporary suspension using commands - PAUSE, WAIT . . . . .	29
7.2	Terminating a program using commands - END . . . . .	29
<b>8</b>	<b>Publication Quality Graphics</b>	<b>29</b>
8.1	Some Commands . . . . .	30
<b>9</b>	<b>GAUSS Procedures</b>	<b>32</b>
9.1	Global and Local variables . . . . .	33
9.2	Naming Conventions . . . . .	33
9.3	Writing Procedures . . . . .	33
9.4	Further Examples . . . . .	35
<b>10</b>	<b>GAUSS Functions and keywords</b>	<b>37</b>
<b>11</b>	<b>Monte Carlo Simulations</b>	<b>38</b>
<b>12</b>	<b>References:</b>	<b>40</b>
12.1	References for this handout . . . . .	40
12.2	Useful GAUSS Resources: . . . . .	40
12.3	GAUSS Resources for Economists: . . . . .	40
<b>13</b>	<b>Appendix: GAUSS Functions and Routines - Quick Reference</b>	<b>41</b>
13.1	Linear Regression . . . . .	41
13.2	Descriptive Statistics . . . . .	41
13.3	Cumulative Distribution Functions . . . . .	41
13.4	Differentiation and Integration Routines . . . . .	41
13.5	Root Finding, Polynomial Multiplication and Interpolation . . . . .	42
13.6	Random Number Generators and Seeds . . . . .	42

# 1 Introduction

GAUSS is a programming language designed to operate with and on matrices. If the core of your task is matrix manipulation in any way, then GAUSS is likely to be a better bet than a lower level programming language. However, GAUSS is not appropriate for, say, writing a menu system; a general-purpose language is probably easier. Nor is GAUSS appropriate for standard applications on standard datasets. There is little point in writing a probit estimation routine in GAUSS for a small dataset. Firstly, there are already routines commercially available for non-linear estimation using GAUSS. More importantly, TSP, LimDep, etc will already perform the estimation and there is no necessity to learn anything at all about GAUSS to use these programs. However, to get extra specification tests, for example, a straightforward solution would be to code a routine and amend the preexisting GAUSS probit program to call the new procedure at the appropriate point in its working.

## 1.1 Running GAUSS in Windows

### 1.1.1 Operating Mode

In the graduate computer lab, open GAUSS with Start - Econometrics Softwares - GAUSS 7.0 - GAUSS 7.0

GAUSS instructions can be carried out in two ways:

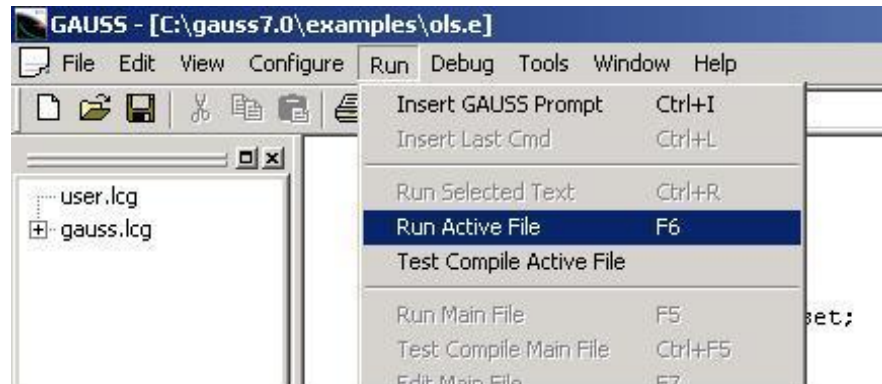
1. **command-line:** this is the dialog window you see in front of you when you start GAUSS. In this mode, commands typed into the GAUSS interface are executed immediately. This allows for an instant response to a command, but the commands cannot be stored. This is therefore not suitable for writing large programs, or for commands which need to be run repeatedly. The place for typing commands is marked by >>.
2. **program or batch:** In this mode, GAUSS commands are typed into a text file. This file is then sent to be GAUSS to be run. This allows one to develop and store complex programs. Example programs are in the directory C:\gauss7.0\examples.

### 1.1.2 Example

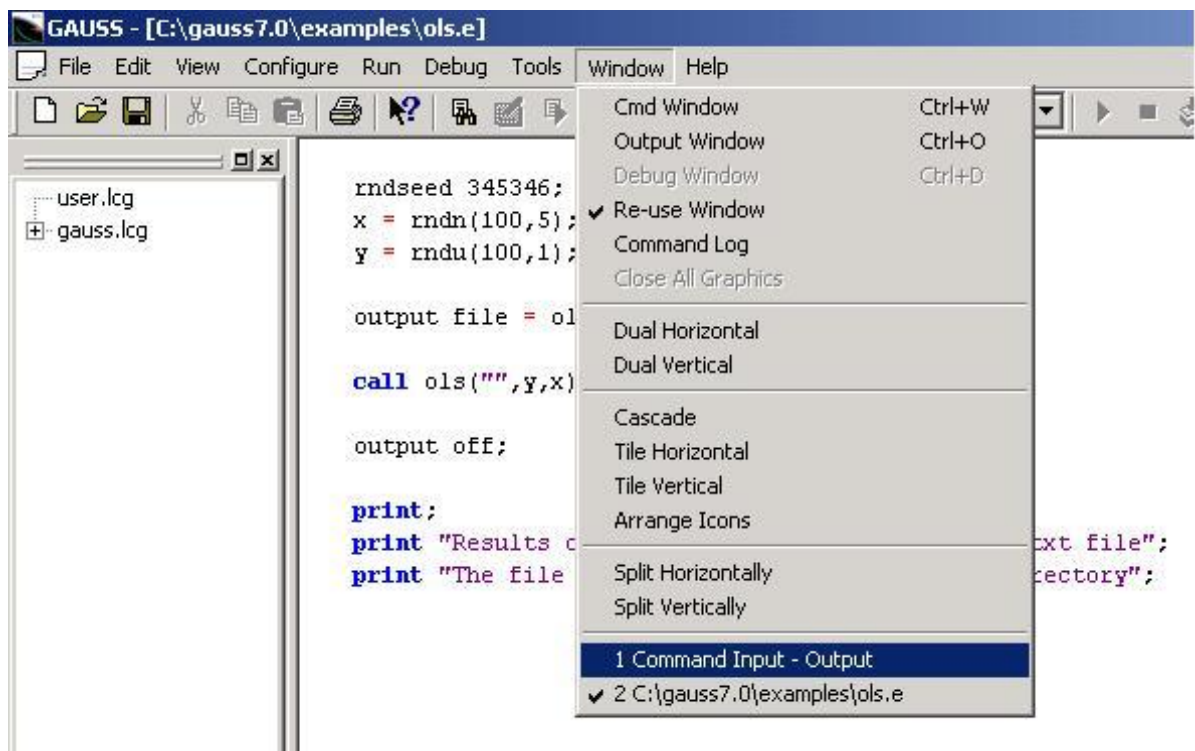
Let's run an example program.

1. In the main menu: File - Open - C:\gauss7.0\examples\ols.e
2. create new folder C:\data
3. In the main menu: File - Save As... - C:\data\ols.e
4. In the main menu: File - Change Working Directory - C:\data

5. Run active file



6. look at results in the Output window



7. note that output is also stored in file C:\data\ols\_output.txt

## 1.2 The Help Menu

GAUSS has two electronic help systems, corresponding to the GAUSS pdf manuals (available at <http://www.aptech.com>).

1. The "*Command Reference*" is an easy way to pick up information on commands (as long as they are not deemed "obsolete"), and is organised both alphabetically and by function, which is useful. It is very terse.
2. The "*User Guide*" is slightly more chatty, and has more examples.

## 1.3 Syntax

- GAUSS is not case-sensitive
- Commands are separated by a semi-colon
- Anything within the `/*...*/` markers is ignored by the program - the markers are used for comments. For example: `/* this is a comment */`.

## 1.4 Note on Numerical Precision

In Gauss, eight bytes are used to store each element of a matrix. Hence, each cell in a matrix can contain up to eight text characters, or numerical data with a range of about  $1.0E \pm 35$ . If you enter text of more than eight characters into the cells in a matrix, the text will be truncated. Numerical data are stored in scientific notation to around 12 places of precision.

## 2 Basics Matrix Operations

### 2.1 Variables

GAUSS variables are of two types: **matrices** and **strings**.

1. **Matrices** may contain numerical data or character data or both.
2. **Strings** are pieces of text of unlimited length. These are used to give information to the user. If you try to assign a string value to an element of the matrix, all but the first eight characters will be lost.

Variables need to have names to reference them. Acceptable names for variables can contain alphanumeric data and the underscore "\_", and must not begin with a number. Reserved words may not be used; standard procedure names may be reassigned, but this is not generally a good idea. Variables names are not case-sensitive.

1. **Acceptable variable names:** eric Eric eric1 eric\_1 \_eric1 \_e\_r\_i\_c
2. **Unacceptable variable names:** 1eric 100 if (reserved word) delif (GAUSS procedure - legal, but foolish)

### 2.2 Creating matrices

1. **Square brackets [ ]** refer to dimensions of a matrix or the coordinates of a block, depending on context. The first number refers to the row, the second the column.
2. **Curly braces { }** are used within GAUSS to group variables together.

New matrices can be defined at any point (except inside procedures). The easiest way is to assign a value to one. There are two ways to do this: (1) by assigning a constant value or (2) by assigning the result of some operation.

#### 2.2.1 Creating a matrix using constant assignment (LET)

The keyword LET creates matrices using constants. Two different forms of creating a matrix (or vector) **x** can be utilized:

#### Specifying the shape of the matrix **x** with the list of constants on the right-hand side

If curly braces are not used, a list of constants separated by space will create a *column vector*.

Adding commas has no effect. Hence the following three operations are equivalent:

<i>Command</i>	<i>Resulting shape of x</i>
LET x = 1 2 3 4 5 6;	Column vector 6x1
LET x = 1,2,3, 4,5, 6;	Column vector 6x1
LET x = 1 2, 3 4, 5 6;	Column vector 6x1

If the list of constants is enclosed in braces {}, then a *row vector* will be produced. Inserting commas in the list of constants instructs GAUSS to form a *matrix*, breaking the rows at the commas.

<i>Command</i>	<i>Resulting shape of x</i>
LET x = {1 2 3 4 5 6};	Row vector 1x6
LET x = {1,2,3, 4,5, 6};	Column vector 6x1
LET x = {1 2, 3 4, 5 6};	Matrix 3x2

**Specifying the shape of the matrix x using square brackets at x[ ]** An  $r \times c$  matrix will be created using the following commands.

The constants on the right-hand side will be allocated to the matrix on a row-by-row basis. Adding commas has no effects.

<i>Command</i>	<i>Resulting shape of x</i>
LET x[3,2] = 1 2 3 4 5 6;	Matrix 3x2
LET x[3,2] = 1, 2, 3, 4, 5, 6;	Matrix 3x2

If only one constant is entered, then the whole matrix will be filled with that number.

<i>Command</i>	<i>Resulting shape of x</i>
LET x[3,2] = 5;	Matrix 3x2

*Note:*

LET x = a\*b;

is **illegal** as "a\*b" is a value and not considered a "constant". In practice, GAUSS will interpret "a\*b" as a string constant and will create a string variable containing the letters and figures "a\*b". The desired outcome is achieved using the syntax described in the following section.

### 2.2.2 Creating a matrix using variable assignment (result of some operation)

The results of any operation can be placed into a matrix without a LET explicit declaration. Assigning a value is done by writing down an equation. The result of the operation

```
m1= m2 + m3;
```

will be that the value "m2+m3" is contained in a variable called "m1". If the variable m1 did not exist before this statement, it will have been created. The size and type of a variable depends entirely on the last thing done with it. Suppose m1 existed prior to the last operation. If m2 and m3 are both scalars, then m1 will now be a scalar - regardless of whether it was previously a matrix, vector, scalar, or string. Variables have no fixed size or type in GAUSS - they can be changed at will simply by assigning a different value to them.

### 2.2.3 Comparison of constant and variable assignment

Why use constant assignments rather than just creating matrices as a result of mathematical or other operations? The answer is that sometimes it is awkward to create matrices of appropriate shapes. It also allows for increased security, as constant assignment checks what values are appropriate and will trap more errors. In practice the main place you will use *constant assignment* will be at the beginning of programs where you set initial values and environment variables (like the name of an output file, or font to use for graphing). During the program you will be using *variable assignment* most of the time and you can ignore the strict rules on constants assignment.

## 2.3 Referencing Matrices

### 2.3.1 Direct Reference

Suppose we have a matrix `mat`. The general format of referencing subsections of `mat` is

```
mat[r1:r2,c1:c2]
```

This will reference a block from row `r1` to row `r2`, and from column `c1` to column `c2` of the matrix `mat`. A value could be assigned to this block; or this block could be extracted for output or transfer to some other location. For example,

```
mat = {1 2 3, 4 5 6, 7 8 9, 10 11 12};  
PRINT mat[2:3,1:2];
```

would print the columns 1 to 2 of rows 2 to 3 of the matrix `mat`:

```
4 5  
7 8
```

To reference only one row or one column, only one coordinate is needed in that dimension:

```
mat[r1,c1:c2]  
or  
mat[r1:r2,c1]
```

For example, to reference the cell in the third row and fourth column of the matrix `mat`, these terms are all equivalent:

```
mat[3:3,4:4]  
mat[3,4:4]  
mat[3:3,4]  
mat[3,4]
```

Entering "." or 0 as a co-ordinate instructs GAUSS to take the whole row or column of the matrix. For example

```
mat[r1:r2,.]
```

means "rows `r1` to `r2` and all columns of matrix `mat`", while

```
mat[0, c1:c2]
```

references for columns `c1` to `c2`.

A whole matrix could then be referred to identically as

```
mat  
or  
mat[.,.]
```

*Note:* This particular feature of GAUSS causes a number of unexpected problems, particularly when using loops to access columns or rows in sequence. If your counter drops to zero (or some unspecified values) then you will find the program operating on all rows or columns instead of just one.

For vectors only one co-ordinate is needed. For example to refer to rows r1-r2 of a column vector `vec` we would type

```
vec[r1:r2]
```

A last way to identify a set of rows or columns is to list them sequentially. For example, to refer to columns 1, 3, and 22 and rows 2 to 4 inclusive of the matrix `mat` we could use

```
mat[2:4,1 3 22]
```

Note that that there are no separating commas in the list of columns; GAUSS treats everything up to the comma as a row reference, everything afterwards as a column reference. If it finds two or more commas within square brackets, it treats this as an error.

These different methods can be combined:

```
mat[1 3:5 9, .]
```

will select every column on rows 1, 3 to 5, and 9. The order is also important:

```
mat[1 2 3, .]
mat[3 2 1, .]
```

will give two matrices with the row order *reversed* in the second one.

### 2.3.2 Indirect Reference

Elements of matrices can also be referred to indirectly. Instead of explicitly using a constant to indicate a row or column number, a variable can also be used. For example,

```
PRINT mat[1:5, .];
and
endRow = 5;
PRINT mat[1:endRow, .];
```

are equivalent.

## 2.4 Reshaping Matrices

RESHAPE is a standard GAUSS function which changes the shape of the matrix. The format is

```
newMat = RESHAPE (oldMat, r, c);
```

where `newMat` is now an  $r \times c$  matrix formed from the elements of `oldMat`. Generally, `newMat` and `oldMat` need to have the same number of elements.

## 2.5 Concatenation

Concatenation refers to joining two (or more) matrices into one. There are two concatenation operators:

~ horizontal concatenation  
| vertical concatenation

These add one matrix to the right or bottom of another. The relevant rows and columns must match. Consider the following operations on two matrices, **a** and **b**, with  $ra$  and  $rb$  rows and  $ca$  and  $cb$  columns. The result placed in the matrix **c**:

dimensions of a	dimensions of b	operation	dimensions of c	condition
$ra \times ca$	$rb \times cb$	$c = a \sim b$	$ra \times (ca + cb)$	$ra = rb$
$ra \times ca$	$rb \times cb$	$c = a   b$	$(ra + rb) \times ca$	$ca = cb$

## 2.6 Matrix Operators

GAUSS has eight mathematical operators and six relational ones. The mathematical operators are:

+ Addition  
- Subtraction  
\* Multiplication  
/ Division  
' Transposition  
% Modulo division  
! Factorial  
^ Exponentiation

The relational operators are:

== EQ equals  
/= NE does not equal  
> GT greater than  
< LT less than  
>= GE greater than/equals  
<= LE less than/equals

Either the symbols or the two-letter acronyms may be used.

Note the double-equals sign for equivalence. This must not be confused with the single-equals sign implying assignment. The two return very different results:

`mat = 5;` mat is assigned the value 5; the "result" of this operation is 5

`mat ==5;` mat is compared to the value 5; the "result" of this operation is "true" if mat is equal to 5, "false" otherwise.

Examples:

```
a = b+c-d;
a = b'*c';
a = (b+c)*(d-e);
a = ((b+c)*(d+e))/((b-c)*(d-e));
a = (b*c)';
```

## 2.7 Conformability and the "dot" operators

GAUSS generally operates in an expected way. If a scalar operand is applied to a matrix, then the operation will be applied to every element of the matrix. If two matrices are involved, the usual conformability rules apply:

Operation	Dimensions of b	Dimensions of c	Dimensions of a
a = b * c;	scalar	$4 \times 2$	$4 \times 2$
a = b * c;	$3 \times 2$	$4 \times 2$	illegal
a = b * c';	$3 \times 2$	$4 \times 2$	$3 \times 4$
a = b + c;	scalar	$4 \times 2$	$4 \times 2$
a = b - c;	$3 \times 2$	$4 \times 2$	illegal
a = b - c;	$3 \times 2$	$3 \times 2$	$3 \times 2$

To tell GAUSS that operations are to be carried out on an "element by element" basis, mathematical (and logical) operators need to be prefixed by a dot:

```
a = b.>c;
a = (b+c).*d';
```

The dot operators do not work consistently across all operands. In particular, for addition and subtraction no dot is needed.

## 2.8 Special matrix operations

### 2.8.1 Some useful matrix types

Three useful matrix creating operations:

```
identMat = EYE (iSize);  identity matrix of size iSize
onesMat = ONES (r,c);    matrix of ones of size  $r \times c$ 
zerosMat = ZEROS (r,c);  matrix of zeroes of size  $r \times c$ 
```

A number of common mathematical operations have been coded in GAUSS. These are simple to use to use and more efficient than building them up from scratch. They are

```
invMat = INV (mat);
invPDMat = INVPD (mat);
momMat = MOMENT (mat, missFlag);
determ = DET (mat);
determ = DETL;
matRank = RANK (mat);
```

The first two of these invert matrices. The matrices must be square and non-singular. `INVDP` and `INV` are almost identical except that the input matrix for `INVDP` must be symmetric and positive definite, such as a moment matrix. `INV` will work on any square invertible matrix; however, if the matrix is symmetric, then `INVDP` will work almost twice as fast because it uses the symmetry to avoid calculation. Of course, if a non-symmetric matrix is given to `INVDP`, then it will produce the wrong result because it will not check for symmetry.

`GAUSS` determines whether a matrix is non-singular or not using another tolerance variable. However, even if it decides that a matrix is invertible, the `INV` procedure may fail due to near-singularity. This is most likely to be a problem on large matrices with a high degree of multicollinearity.

The `MOMENT` function calculates the cross-product matrix from `mat`; that is, `mat'*mat`. For anything other than small matrices, `MOMENT(x, flag)` is much quicker than using `x'x` explicitly as `GAUSS` uses the symmetry of the result to avoid unnecessary operations. The `missFlag` instructs `GAUSS` what to do about missing values - whether to ignore them (`missFlag=0`) or excise them (`missFlag=1` or `2`).

`DET` and `DETL` compute the determinants of matrices. `DET` will return the determinant of `mat`. `DETL`, however, uses the last determinant created by one of the standard functions; for example, `INV`, `DET` itself, decomposition functions all create determinants along the way. `DETL` simply reads this value. Thus `DETL` can avoid repeating calculations. The obvious drawback is that it is easy to lose track of the last matrix passed to the decomposition routines, and so determinants should be read as soon as possible after the relevant decomposition function has been called.

`RANK` calculates the rank of `mat`.

### 2.8.2 Manipulating matrices

There are a number of functions which perform useful little operations on matrices. Commonly-used ones are:

```
vec = DIAG (mat);
mat = DIAGRV (vec);
newMat = RESHAPE (oldMat, newRows, newCols);
nRows = ROWS (mat);
nCols = COLS (mat);
maxVec = MAXC (mat);
minVec = MINC (mat);
sumVec = SUMC (mat);
```

`DIAG` and `DIAGRV` abstract and insert, respectively, a column vector from or into the diagonal of a matrix.

`ROWS` and `COLS` return the number of rows and columns in the matrix of interest.

`MAXC`, `MINC`, and `SUMC` produce information on the columns in a matrix.

`MAXC` creates a vector with the number of elements equal to the number of columns in the matrix. The elements in the vector are the maximum numbers in the corresponding columns of the matrix. `MINC` does the same for minimum values, while `SUMC` sums all the elements in the column. However, note that all these functions return column vectors. So, to concatenate onto the bottom of a matrix the sum of elements in each column would require an additional transposition:

```
sums = SUMC(mat1);
mat1 = mat1 | sums';
```

On the other hand, because these functions work on columns, then calling the functions again on the column vectors produced by the first call allows for matrix-wide numbers to be calculated:

```
maxMat=MAXC(MAXC(mat1));
minMat=MINC(MINC(mat1));
sumMat=SUMC(SUMC(mat1));
```

will return the largest value in mat1, the smallest value, and the total sum of the elements.

### 2.8.3 Example 1

```
new;
x = {2 5,0.9 11};
y = {0.5 0.2,3 4};
z = y.*x;
v = x.^y;
x*~y;
print z;
print v;
```

### 2.8.4 Example 2

```
new;
let x[2,2] = 1 0 2 0;
let y[2,2] = 1 5 0 0;
z = .NOT x;
print z;
z = x.and y;
print z;
z = ( ( .NOT x) .XOR y ) .OR Y;
print z;
```

### 2.8.5 Example 3

```
new;
let x[2,2] = 1 0 2 0;
let y[2,2] = 1 5 4 2;
z = x > y;
print z;
z = x <= y;
print z;
z = x == y;
print z;
z = x .== y;
print z;
z = x ./= y;
print z;
```

```
z = (x ./=y) .AND (x.==y);  
print z;
```

#### 2.8.6 Example 4

```
new;  
x = seqa(1,1,4);  
y = x'*ones(10,4);  
indx = {1 3 4};  
z = y[.,indx];  
print z;
```

#### 2.8.7 Example 5

```
new;  
y = rndn(100,4);  
ex = y[.,2] .> 0.5;  
suby = selif(y,ex);  
print suby;  
ex2 = abs(y[.,1]) .< 0.33 .AND y[.,2] .> 0.66;  
suby2 = selif(y,ex);  
print suby2;  
ex = y[.,1] .< 0.33;  
y = delif(y,ex);
```

## 3 Input and output

GAUSS handles data on disk in a number of formats. It can read and create standard text files and older spreadsheet formats, as well as using its own format to store matrices, datasets or code samples.

### 3.1 Loading Gauss Data with `loadd`

`loadd` loads a GAUSS dataset in its entirety. The following command loads the GAUSS dataset *autoreg.dat*:

```
m = loadd("c:\\gauss7.0\\examples\\autoreg");
```

### 3.2 Using the Matrix Editor

The Matrix Editor lets you view and edit matrix data in your current workspace. You can open the Matrix Editor from either the Command Input - Output window or a GAUSS edit (program) window by highlighting a matrix variable name and typing *Ctrl+E*. You can view multiple matrices at the same time by opening more than one Matrix Editor. The Matrix Editor will also allow you to format matrices.

Alternatively, you can view a matrix by changing the left-hand side bar to "*Symbols*" and double-clicking the matrix.

### 3.3 Text files (.txt)

#### 3.3.1 ASCII input

Input can be taken from ASCII (i.e. normal alphanumeric text) files using the `load` command. This is augmented by the addition of square brackets which indicate the ASCII nature of the file:

```
LOAD varName[] = fileName;  
LOAD varName[r, c] = fileName;
```

In the first case, GAUSS will load the contents of `fileName` into the column vector `varName`, which can then be checked for size and reshaped. This is the preferred option for loading ASCII files. Items can be numeric or text and should be separated by spaces or commas. Line breaks are treated as white space: GAUSS does not use them to distinguish rows. Text items longer than eight characters will be truncated.

The second form loads the file into an `r` by `c` matrix. If there are too many elements in the file for the matrix, then the extra ones will not be read; if the file does not contain enough data items, then the ones found will be repeated until the matrix is full.

#### 3.3.2 ASCII output and writing on the screen

Producing ASCII output files is no different from displaying on the screen. GAUSS is treating the output as something to be "displayed" (even if only to a file). Thus, GAUSS allows for all output to be copied and redirected to a disk file. Anything which appears on the screen can also appear in the disk file. To produce an ASCII file therefore requires that

1. an output file is opened
2. PRINT is used to display all the information to go into the output file
3. the output file is closed when no more output is to be sent to it.

The relevant command to begin this process is **OUTPUT**:

```
OUTPUT FILE = fileName ON;
OUTPUT FILE = fileName RESET;
```

Both will instruct GAUSS to send a copy of everything it displays, from that point onward, to the file `fileName`.

If `fileName` does not already exist, then these two are identical; but if the file does exist, then the first form ensures that any output is *appended* to the existing contents of the file, while the second *empties* the file before GAUSS starts writing to it.

If no file name is given, then GAUSS will use the default "output.out". There is no default extension for output files.

Once a file has been opened, it can be closed and opened any number of times by combining the above commands with

```
OUTPUT OFF;
```

These commands will all work on the last recorded file name given. The `FILE=fileName` bit could be included here as well if the user wishes to swap between different output files; generally, however, only one output file is used for a program, and so naming the file explicitly is superfluous. An analogous command `SCREEN` switches screen output on and off. These two commands are independent and so screen display off and file output on is a perfectly acceptable combination.

**Example 1** sends output to one file only:

```
OUTPUT FILE="eric.txt" RESET;
:
OUTPUT OFF:
:
OUTPUT ON;
:
OUTPUT OFF
:
OUTPUT ON;
:
```

**Example 2** sends output to two different files, "eric1.txt" and "eric2.txt":

```
OUTPUT FILE= "eric1.txt" RESET;
:
OUTPUT OFF:
:
OUTPUT FILE="eric2.txt" RESET;
:
OUTPUT OFF
:
OUTPUT FILE="eric1.txt" ON;
:
```

### 3.4 Spreadsheets

GAUSS can import data directly from Excel spreadsheets. Using the import and export functions is much more convenient than using ASCII files as intermediaries, as well as being more reliable. For reading files:

```
mat = SPREADSHEETREADM(fileName,range,sheet);
```

#### Input

1. file string, name of .xls file.
2. range string, range to read or write; e.g., "a1:b20".
3. sheet scalar, sheet number.

#### Output

xlsmat matrix of numbers read from Excel. The import command just returns a matrix and it's up to the user to break off row or column headings.

For writing files:

```
okay = SPREADSHEETWRITE(mat,fileName,range,sheet);
```

#### Input

1. data matrix, string or string array, data to write.
2. file string, name of .xls file.
3. range string, range to read or write; e.g., "a1:b20".
4. sheet scalar, sheet number.

#### Output

xlsmat success code, 0 if successful, else error code.

### 3.4.1 Example

```
new;
y = seqa(1,1,10);
x = rndu(10,1);
data = y~x;
filename = "c:\\gauss50\\eser\\gauss2excl.xls";
range = "a1:b10";
ret = spreadsheetwrite(data,filename,range,1);
ret;
filename = "c:\\gauss50\\eser\\gauss2excl2.xls";
range = "a1:b11";
xlsmat = SpreadsheetReadM(filename, range, 1);
printfm(xlsmat,zeros(1,2)|ones(10,2),fmt);
```

## 3.5 Storing matrices (.fmt files)

GAUSS stores matrices in files with a `.fmt` extension. This is the default option - if no extension is given to file names, GAUSS will assume it is reading or writing a matrix file.

The commands for matrix files are

```
LOAD varName = fileName;
SAVE fileName = varName;
```

`varName` is the name of the variable in memory to be saved or loaded.; `fileName` is the name of the matrix file with no `.fmt` extension. For example,

```
SAVE "file1" = mat1;
LOAD mat2 = "file1";
```

creates a file on disk called `file1.fmt` which contains the matrix `mat1`. This is then read into a new matrix, `mat2`.

Note: there are numerous variations on the basic `LOAD` command (see the GAUSS Command Reference for details).

## 3.6 Datasets (.dat files) In Depth

Unlike the GAUSS matrices, reading from or writing to a GAUSS dataset is not a single, simple operation. For matrices, the whole object is being moved into memory or onto disk. By contrast, a GAUSS dataset is used in a number of stages:

1. the file must be opened
2. then it may be read from or written to, which may involve the whole file or just a few lines
3. when references to the file are finished, it should be closed.

All files used will be given a *handle* by GAUSS; this is a scalar which is GAUSS's internal reference for that file. It will be needed for all operations on that file, and so should not be altered. The handle is needed because several files can be 'open' at one time (for example, reading from one, writing to another); precisely how many depends on the computer's configuration. Without the file handle, a dataset cannot be accessed, and if the file handle is overwritten then the wrong file may be used.

### 3.6.1 Opening datasets

A dataset must be opened for either reading or writing or "updating" (both). Once a dataset has been opened for one "mode" it cannot be switched to another. The command is

```
OPEN handle = fileName FOR mode VARINDXI offset;
```

*handle* is a non-negative scalar, the file handle returned to you if the operation is successful (if the command did not work, the handle is set to -1). *The file handle should always be set to zero before this command, to avoid the possibility of GAUSS trying to open a file already open.* *fileName* is as above.

*mode* is one of READ, APPEND, or UPDATE. If the mode is omitted, GAUSS defaults to READ. If READ is chosen, updating the file is not allowed. Choosing APPEND means that data can only be appended to the file; the existing contents cannot be read. UPDATE allows reading and writing. When GAUSS opens the file with VARINDXI, it reads the names of fields (columns) and prefixes them all with "i" (for index). These can then be used to reference the columns of the dataset symbolically instead of using column numbers explicitly. This makes programs more readable, more easily adapted, and less likely to be upset by changes in the structure of the dataset. In the above example, the four columns in the dataset created could be referred to as 1 to 4 or, equivalently but much more usefully, as *iname*, *iage*, *isex*, *iwave*. Using these index variables without VARINDXI causes some problems for GAUSS when it is checking a program prior to running it, so although VARINDXI is optional it should generally be included.

The *offset* scalar option shifts all these indexes by a scalar and so is useful if the data is to be concatenated horizontally to another matrix or dataset. However, usually it can be left out. As an example, to open the file created in the previous sub-section for reading, the command would be

```
OPEN handle1 = "file1" FOR READ VARINDXI;
```

which would give a file handle in *handle1*, and four scalar indexes: *iname*, *iage*, *isex*, and *iwave*, set to 1, 2, 3, and 4 respectively.

### 3.6.2 Creating new datasets

To start a new dataset for writing, it must be created. This is done by

```
CREATE handle = fileName WITH colNames, columns, type;
```

*handle* is the handle GAUSS will return if it is successful in creating *fileName*.

*fileName* may be a *constant* like "file1", or it may be a *string*, referenced using the ^ operator.

*colNames* is the list of names for the columns (usually a character vector)

*columns* tells GAUSS how many columns of data there are (which is not necessarily the same as the number of names - it may be sensible to have some "spare" columns)

`type` is the storage precision of the data - integers, single precision, or double precision. For example,

```
fileName = "file1";
varNames = "Name" "age" "sex" "wage";
CREATE handle1 = ^fileName WITH ^varNames, 4, 4;
```

prepares a datafile called `file1.dat` for writing. A header file `file1.dht` will also be created, which records that the datafile should contain four columns, named "Name", "age", "sex" and "wage", and in single precision (`type=4`, the default).

When a file is `CREATED`, it is automatically opened in `APPEND` mode (obviously; there is nothing to be read as yet). However, creating new datasets is much rarer than accessing a preexisting dataset, and so `OPEN` is more common than `CREATE`.

### 3.6.3 Reading datasets

A GAUSS dataset is explicitly composed of rows of data, and these rows are the basic unit of manipulation. One or more rows is read at a time and data is parcelled up into rows before being written.

To **read** data, the command is

```
dataMat = READR (handle, numLines);
```

which reads `numLines` rows from the file referenced by `handle` into the data matrix `dataMat`. Rows and columns in the dataset become rows and columns in the matrix. So, in our above example,

```
dataMat1 = READR (handle, 10);
```

reads ten lines from the dataset and creates a 10x4 matrix called `dataMat1` which can be accessed like any other variable.

Attempting to read past the end of the file will cause the program to crash.

### 3.6.4 Writing datasets

**Writing** data is just the reverse. The command

```
result = WRITER (handle, dataMat);
```

will try to add `dataMat` into the file at the current file position. `dataMat` must have the same number of columns as the data currently in the file, or GAUSS will fail. Data in the dataset will be overwritten.

`result` is the number of lines actually written to disk.

### 3.6.5 Closing datasets

Files should always be closed when reading or writing is finished. GAUSS will automatically do this when leaving the GAUSS environment or when it encounters an `END` statement (see later). However, having files open unnecessarily may slow the system down; may prevent new (and useful) files being opened; may be mistakenly altered by the program; and may be corrupted or lose data due to system failure.

Files are closed by the `CLOSE` command:

```
result = CLOSE (handle);
```

If the file for `handle` was closed successfully, then `result` will be set to 0; otherwise, it will be -1. The reason the `handle` is set to 0 on success and -1 on failure is because valid handles are all positive numbers; therefore, GAUSS uses zero and negative numbers to indicate the state of the file handle. If the `CLOSE` worked, then `handle` should be set to zero, to signify that there is no open file attached with this handle (this information is used by `OPEN` and `CREATE`). This could be combined by using

```
handle = CLOSE (handle);
```

as recommended by the GAUSS manual. An alternative is to use one of the following:

```
CLOSEALL;  
CLOSEALL handle1, handle2, ... handlex;
```

which closes all or a specified list of files. The first form does not set file handles to zero. The second form sets handles to zero, but GAUSS is silent on the possibility of the closure failing.

## 4 Managing data

### 4.1 SHOW, PRINT, FORMAT, NEW, CLEAR, DELETE

#### 4.1.1 SHOW

SHOW displays the name, size and memory location of all global variables (explained later) and procedures in memory at any moment.

```
SHOW varName;  
or  
SHOW/m varName ;
```

where `varName` is the variable of interest.

The "wild card" symbol "\*" can be used, so that

```
SHOW er* ;
```

will find all references beginning with "er". The /m parameter means that only matrices are displayed.

#### 4.1.2 PRINT

PRINT displays the contents of matrices and strings. The format is

```
PRINT var1 var2 var3... varx ;
```

which prints the list of variables. How it prints depends on the data. If the data fits on one line (all row vectors, scalars, or strings) then PRINT will display one after the other on the same line. If, however, one of the variables is a matrix or column vector, then the variable immediately following the matrix will be printed on a new line.

Strings need to be inserted in two apostrophes:

```
PRINT "Hello";
```

#### 4.1.3 FORMAT

PRINT style is controlled by the FORMAT commands, which sets the way matrices (but not strings) are printed. There are options to print numbers and character data with varying field widths, decimal expansion, justification, spacing and punctuation. These are covered in the manual and are all similar in form to:

```
FORMAT /RD 6, 0;
```

where, in this case, we have numbers right-justified (/RD), separated by spaces (/RDC would do commas), with 6 spaces left for writing the number and 0 decimal places. If the number is too large to fit into the space, then the field will be expanded but for that number only - not the whole matrix. Strings are given as much space as they need, but no spaces are inserted between them. The print styles set by FORMAT operate from the time they are set until the next FORMAT command is received.

#### 4.1.4 NEW, CLEAR, and DELETE

These three all clean up memory. They do not affect files on disk. `NEW` clears all references from memory. It can be called from inside a program, but obviously this is rarely a smart move. The exception is at the start of a program. A call to `NEW` will remove any junk left over from previous work, leaving all memory free for the new program. `NEW` has no parameters and is called by

```
NEW;
```

Calling `NEW` at the start of a program ensures that the workspace is cleared of unwanted variables, and is good practice. Calling `NEW` at any other point is usually disastrous and not so highly recommended.

`CLEAR` sets particular variables to zero, and it can also be called by a program. It is useful for tidying up data and initialising variables:

```
CLEAR var1 var2 ... varN ;
```

Because it sets the variable to the scalar zero, then `CLEAR` is identically equal to a direct assignment:

```
CLEAR x; is equivalent to x = 0;
```

`DELETE` clears variables from memory, and so is a better option than `CLEAR` for tidying up unwanted variables. However, it cannot be called from inside a program. The delete command is like `SHOW`:

```
DELETE varName;  
DELETE/n varName;
```

where `varName` can include the wild card character. The `/n` option stops `GAUSS` double-checking the deletion is wanted. The special word "ALL" can be used instead of `varName`; this deletes all references, and so

```
DELETE/N ALL;
```

is equivalent to `NEW`.

## 5 Statistical Functions

Here are some of the basic statistical functions:

C = Corrx(X)    Correlation matrix of matrix X  
M = Meanc(X)    Means of columns  
S = Stdc(X)    Standard deviation of matrix X's columns  
V = Vcx(X)    Variance-covariance matrix X

### 5.1 Example 1

```
rndseed 5390;
p =100;
q = 10;
x = rndn(p,q);
mu = meanc(x);
st = stdc(x);
vc = vcx(x);
print "sample mean " mu;
print "sample variance" st.^2;
print "variance-covariance matrix";
print;
vc;
```

### 5.2 Example 2

Compute the correlation coefficient

$$\rho = \frac{\frac{1}{N} \sum_{t=1}^N (x_t - \bar{x})(y_t - \bar{y})}{\left[ \frac{1}{N} \sum_{t=1}^N (x_t - \bar{x})^2 \right]^{1/2} \left[ \frac{1}{N} \sum_{t=1}^N (y_t - \bar{y})^2 \right]^{1/2}}$$

```
new;
rndseed 4572;
u = rndu(1000,1);
x = 10 + 2*u;
y = x.*lag1(x).*lagn(x,2).*lagn(x,3);
y = trimr(y,3,0);
x = trimr(x,3,0);
yc = y - meanc(y);
xc = x - meanc(x);
covxy = meanc(xc.* yc);
sigmax = sqrt(meanc(xc .* xc));
sigmay = sqrt(meanc(yc.*yc));
corrxy = covxy ./ (sigmax*sigmay);
print corrxy;
corrxy(y~x);
```

## 6 Flow of Control

There are two main ways of changing the flow of control of the program: the conditional branch and the loop.

### 6.1 Conditional branching: IF

the simplest IF statement is:

```
IF condition1;
    doSomething1;
ENDIF;
```

There are two optional conditioning statements ELSEIF and ELSE.

```
IF condition1;
    doSomething1;
ELSEIF condition2;
    doSomething2;
ELSEIF condition3;
    ...
ELSE;
    doSomething4;
ENDIF;
```

- 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.
- If several conditions are "true", then GAUSS will act on the first true condition found and ignore the rest.
- The ELSE section has no associated condition; therefore, if GAUSS reaches the ELSE statement it will always execute the ELSE section. To reach the ELSE, GAUSS must have found all other conditions "false". So, ELSE is a catch-all category: it is only called when no other conditions are met, but if the ELSE section is included then some action will always be taken.

**Example:**

```
x = {1 3 26, -1 3 567};
y = x < 0;
print y;
z = x.< 0;
w = x < 1000;
print w;
a = {1 2};
b = {2 0};
if a < b;
    print "true";
```

```

else;
    print "false";
endif;
if a < 2;
    print "true";
else;
    print "false";
endif;

```

## 6.2 Loop statements: DO WHILE/UNTIL and FOR

GAUSS has two types of loops:

1. FOR loops: used when the number of loops is fixed and known in advance;
2. DO WHILE/UNTIL loops: used when the conditions to enter or exit the loop need to be continually re-evaluated.
  - (a) DO WHILE loops until condition is "false"
  - (b) DO UNTIL loops until condition is "true".

The condition is tested before the loop is entered; therefore the loop might not be entered at all.

*Note: The usage of the following two functions is **not recommended**. They are listed here only for the sake of completeness. Use IF statements to ensure an orderly exit from a loop; it makes the program much more traceable.*

Two functions, BREAK and CONTINUE, allow the cycle to be interrupted:

- BREAK breaks out of a DO or FOR loop.
- CONTINUE jumps to the top of a DO or FOR loop.

### 6.2.1 FOR loops

A FOR loop cycles a fixed number of times. The format of the FOR loop is

```

FOR i(start, stop, interval);
:
ENDFOR;

```

- The variables **start**, **stop**, and **interval** control the number of times the loop operates. The loop will count from start to stop in steps of interval.
- The counter **i** can be referenced within the loop, but should not be changed by another command reference. It can take only integer values.

**Example 1:** display: 10 9 8 7 6 5 4 3 2 1

```
FOR i (10, 1, -1);
  PRINT i;;
ENDFOR;
```

**Example 2:** take a sum of all ODD numbers between 0 and 100

```
sum = 0;
FOR i (1,99,2);
  sum = sum + i;
ENDFOR;
PRINT "The sum of the first 50 odd numbers:";
PRINT sum;;
```

### 6.2.2 DO WHILE/UNTIL loops

The format for the DO WHILE and DO UNTIL loop statement is

```
DO WHILE condition;
  doSomething;
ENDDO;
```

```
DO UNTIL condition;
  doSomething;
ENDDO;
```

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;
```

and

```
DO UNTIL (NOT condition);
```

are identical.

**Example 1:** display: 10 9 8 7 6 5 4 3 2 1

```
i = 10;
DO WHILE i /=0;
  PRINT i;;
  i = i - 1;
ENDDO;
```

In the example above, note that the condition is set before entering the loop, and it needs to be updated explicitly. If the line "*i = i - 1;*" was not included, then *i* would have stayed at 10, the

condition would not have been met, and the program would have continued printing out "10" forever.

**Example 2: take a sum of all squared numbers 1 to 100**

```
sum = 0;
i = 1;
do until i > 100;
    sum = sum + i^2;
    i = i + 1;
endo;
```

**Example 3: verify if each element of a matrix is greater than those on the boundary**

```
new;
x = rndn(20,10);
x = x.*x;
format 7,3;
Print "Generated Matrix";
print x;
print;
print;
icol = 2;
format 3,3;
do while icol < cols(x);
    irang = 2;
    do while irang < rows(x);
        if x[irang-1:irang+1 , icol-1:icol+1] >= x[irang,icol];
            print "x[ " irang ", " icol " ] satisfies condition";
        endif;
        irang = irang + 1;
    endo;
    icol = icol + 1;
endo;
```

## 7 Suspending execution

All these commands stop execution either temporarily or permanently. In addition, some key combinations may stop a program in an emergency.

### 7.1 Temporary suspension using commands - PAUSE, WAIT

Three commands can lead to the temporary suspension of a program:

<code>PAUSE(sec);</code>	will wait for <code>sec</code> seconds before the program continues
<code>WAIT;</code>	will wait until a key has been pressed
<code>WAITC;</code>	will clear the keyboard buffer before waiting for a key so that the program will always stop long enough.

These functions are most useful where the program is stopped while something is being checked or a message is displayed which should be read. `WAIT` and `WAITC` cannot be used to read console input.

### 7.2 Terminating a program using commands - END

Ideally, a program should close all files and reset all screen and output options before it terminates. The command

```
END;
```

will carry out these functions. `END` tells GAUSS that the program is complete. Even if there are more instructions, the program will terminate at this point. `END` can be placed anywhere in a program. Whenever it is encountered, the program stops.

## 8 Publication Quality Graphics

GAUSS *Publication Quality Graphics* consists of a set of main graphing procedures and several additional procedures and global variables for customizing the output.

There are four basic parts to a graphics program:

#### 1. Header

In order to use the graphics procedure, the *pgraph* library must be active. This is done in the library statement at the top of the program.

```
library pgraph;  
graphset;
```

`Graphset` resets the graphical global variable to the default state.

#### 2. Data setup

The data to be graphed must be in matrices.

#### 3. Graphics format setup

Most of graphics elements contain defaults which allow the user to generate a plot without modification. Then defaults may be overridden by the user through the use of global variables and graphics procedures. The variables that begins with ”\_p” are the global control variables used by the graphics routines.

#### 4. Calling Graphics Routines

It is the main graphics routines where all of the work for the graphics functions get done. The graphics routines takes as input the user data and global variables which have previously been set up.

### 8.1 Some Commands

This section serves for illustrative purposes only. For a complete reference on graphics, consult more elaborate GAUSS resources. An excellent reference is Eric Zivot's webpage at <http://faculty.washington.edu/ezivot/pqg.htm>

XY (x, y)	XY plot
HIST(x, y)	Computes and graphs a frequency histogram for a vector
BOX(grp, y)	Graphs data using the box graph percentile method
SURFACE(x, y, z)	Graphs a 3-D surface
CONTOUR(x, y, z)	Graphs a matrix of contour data
XLABEL(str)	To set a label for the X axis
Y LABEL(str)	To set a label for the Y axis
ZLABEL(str)	To set a label for the Z axis
TITLE(str)	To set the title for the graph

#### Example program:

This program is a simple OLS estimation procedure for the model

$$\log(wage) = \beta_0 + \beta_1 married + \beta_2 educ + u$$

in Wooldridge's *Econometric Analysis of Cross Section and Panel Data*, exercise 4.1.

The data can be accessed at

<http://www.msu.edu/~ec/faculty/wooldridge/book2.htm>

Data files – text file 4

We will first read the data into Excel, save them in Excel format and use `SpreadsheetReadM` to read them into GAUSS. This is arguably the simplest and safest input method.

To obtain the estimation results, follow these steps:

1. From the zip file, extract *nls80.raw* (contains data) and *nls80.des* (contains description of the data).
2. Copy into C:\gauss6.0\tmp
3. Open Excel
4. In Excel open the file *nls80.raw* - don't forget to change the file type to "all files" (\*,\*)
5. Save into C:\gauss6.0\tmp as an Excel workbook *nls80.xls*
6. Save the program below into C:\gauss6.0\tmp\tryols.prg
7. In the GAUSS command line menu, type: `run C:\gauss6.0\tmp\tryols.prg`

```

/* program tryols.prg */
new;
xlsmat = SpreadsheetReadM("C:\\gauss6.0\\tmp\\nls80.xls", "a1:q935", 1);
x = xlsmat[., 9 5];
y = log(xlsmat[., 1]);
n=rows(x); k=cols(x);
const=ones(n,1);
x1=const~x;
beta_hat=invpd(x1'x1)*(x1'y);
y_hat=x1*beta_hat;
u_hat=y-y_hat;
u1=u_hat.*x1;
SST=(y-meanc(y))'(y-meanc(y));
SSE=(y_hat-meanc(y_hat))'(y_hat-meanc(y_hat));
SSR=u_hat'u_hat;
sig2_hat=SSR/(n-k-1);
se_bhat=sqrt(diag(invdpd(x1'x1)*sig2_hat));
seh_bhat=sqrt(diag(invdpd(x1'x1)*(u1'u1)*invdpd(x1'x1)));
R2=1-SSR/SST;
R2adj=1-sig2_hat/(SST/(n-1));
xname="constant";
i=1; do while i<=k;
xname1="" $+ "x" $+ ftocv(i,1,0) $+ " ";
xname=xname|xname1;
i=i+1; endo;
hname="Variable"~"Estimate"~" SE "~-HCSE ";
ff = "#*.1G"~10~5;
ffn="#*.1G"~10~8;
@ format /LZ 10,4;@
"";
"OLS Estimation Results";
"=====";
pr=printfm(hname,0~0~0~0,ffn);"";
"-----";
pr=printfm(xname~beta_hat~se_bhat~seh_bhat,0~1~1~1,ffn|ff|ff|ff);
"=====";
"number of observations " n;
"R_squared " R2;
"R_squared (adjusted) " R2adj;
"residual sum of squares (SSR) " SSR;

```

## 9 GAUSS Procedures

Procedures are self-contained blocks of code. When they are called by the program, the chain of command within the program switches to the procedure; when the procedure has completed all its operations, control returns to the main program.

A procedure is called with some parameters, something happens, and a result may be returned. The parameters may be constants or variables; any returned values must be placed in variables. There may be any number of input and output parameters, including none. The general format is

```
{outVar1, ...outVarN} = ProcName(inVar1, ... inVarN);
```

- the `inVar` parameters are giving information to the procedure
- the `outVar` variables are collecting information from the procedure
- curly brackets `{}` to group variables together for the purposes of collecting results
- round brackets `()` to delineate the input parameters

If there is one or no parameter, then the form can be simplified:

```
{outVar1, ... outVarx} = ProcName (inVar);    one input parameter  
{outVar1, ... outVarx} = ProcName;          no input parameter  
ProcName (inVar1, ... inVarx);              no returned result  
outVar = ProcName (inVar1, ...inVarx);      one result returned
```

For example, the procedure `DELIF` requires two input parameters (a matrix and a column vector), and returns one output, a matrix:

```
outMat = DELIF(inMat, colVec);
```

The procedure `EIGCG` requires two input parameters and two output parameters

```
{eigsReal, eigsImag} = EIGCG(matReal, matImag);
```

The procedure `SORT` needs four input parameters but returns no result:

```
SORT(inFile, outFile, keyName, keyType);
```

*Note:* For all procedures, it is the programmer's responsibility to ensure that the right sort of data is used. If a procedure is expecting a scalar as a parameter and you pass it a row vector, for example, this will not be flagged as an error when GAUSS checks the program syntax. It may or may not cause the procedure to crash but this will not be apparent until the program is running. All GAUSS will check is that the correct number of parameters is being passed back and forth.

## 9.1 Global and Local variables

- A variable always has a certain *scope*. A scope is the part(s) of the program in which the variable is visible (=accessible) by the computer. All of the variables considered so far have been global: they are visible to all parts of the program.
- Procedures allow the use of local variables: they can only be seen within the ambit of the procedure. Anything outside that procedure cannot read or access those variables; as far as the program outside the procedure goes, that variable does not exist.
- Local variables are only visible at the level at which they were declared. Procedures may be nested: one procedure may call another. However, the local variables are only visible to those procedures in which they were called: they are not visible to procedures they call or were called by. Local variables only exist for the life of the procedure; once the procedure is completed and control returns to the calling code, all variables local to that procedure will be deleted from memory. If the procedure is called again, the local variables will be a completely new set, not the set that was used last time the procedure was called. Obviously, local variables always start off uninitialised.
- Global variables cannot be declared inside a procedure. They may be used, their size may be changed, but they may not be declared afresh. Any variable which is used in a procedure must be either declared explicitly as a local variable or be a preexisting global variable.

## 9.2 Naming Conventions

Because procedures cannot see the variables created by other procedures, variables with the same name can be used in any number of procedures. If, however, variable names do conflict, (a global variable has the same name as a local variable), then the local variable always takes precedence.

## 9.3 Writing Procedures

A procedure definition consists of five parts:

1. Procedure declaration: **proc** statement
2. Local variable declaration: **local** statement. These are variables that exist only when the procedure is executing. They cannot conflict with other variables of the same name in your main program or in other procedures.
3. Body of procedure
4. Return from procedure: **retp** statement
5. End of procedure definition: **endp** statement

General syntax structure of a procedure:

```
PROC (numReturns) = ProcName( inParam1, inParam2,... inParamN);  
LOCAL locVar1;  
:  
LOCAL locVarN;  
    instruction1;
```

```

    instruction2;
    :
    instructionN;
RETP(outParam1, outParam2, ... outParamN);
ENDP;

```

Example: write a procedure that returns  $\sqrt{x} + x^{-1}$  given the input variable  $x$ :

```

proc(1) = sqrtinv(x); /* procedure declaration */
local y;             /* local variable declaration */
    y = sqrt(x);     /* body of procedure */
retp(y+inv(x));     /* return from procedure */
endp;                /* end of procedure */

```

The procedure can be called from the main program with

```
z=sqrtinv(x);
```

- The input parameter  $x$  is a variable which can be used like any other. It is a copy of the variable with which the procedure was called. Therefore it can be altered in any way inside the procedure and this will have no effect on the original variables.
- Local variables are declared using the `LOCAL` statement. Any variables used in the procedure which are not input parameters or global variables must be declared here. Variables can be defined in two ways:

```

LOCAL x;
LOCAL y;
LOCAL z;

```

or

```
LOCAL x, y, z;
```

- Note that there is no information about the size or type of the variable here. All this statement says is that there are variables  $x$ ,  $y$ , and  $z$  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.
- If there is no value to be returned, then the `RETP` statement can be omitted.
- 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" in any meaningful sense when it is called; otherwise it is just a definition.

## 9.4 Further Examples

### 9.4.1 Example 1

Consider this simple procedure to take a column vector and fill it with ascending numbers. The start number and increment are given as parameters. This mimics the action of the standard function `SEQA`:

```
PROC(i) = FillVec(inVec, startNum, step);
LOCAL i;
LOCAL nRows;
    nRows = ROWS (inVec);
    inVec[1] = startNum;
    i = 1;
    DO WHILE i <= nRows;
        inVec[i] = inVec[i-1] + step;
        i = i + 1;
    ENDO;
RETP(inVec);
ENDP;
```

This procedure could be called from the main program (or other procedure) by, for example,

```
sequence = FillVec(ZEROS(10, 1), 10, 10);
```

which would give a  $(10 \times 1)$  vector counting to one hundred in tens. In this case, even though the parameters are variables within the procedure, they were created using constants. This is due to the fact that parameters are copies of the variables passed to the procedure. In the above example, `GAUSS` calculated the results of the `ZEROS` operation; created three new variables, `inVec`, `startNum`, and `step`, which have no further connection to the original values `ZEROS(..)`, `10`, `10`; and then made these new variables visible to `FillVec`, and `FillVec` only. Thus to concatenate an index vector onto an existing matrix, a program could use

```
temp = FillVec(mat[:,1], 1, 1);
mat = mat ~temp;
```

or, equivalently and without needing an extra variable,

```
mat = mat ~FillVec(mat[:,1], 1, 1);
```

The column of `mat` used as the input vector will not be altered by the procedure call. Note that when a procedure returns a single result, it can be treated like the result of any other operation. Thus, given a vector `iVec`, a valid command could be

```
result = SQRT((FillVec(iVec, 50, 1).*FillVec(iVec, 50, -1))*ONES(50,1));
```

A procedure is called the same way as an intrinsic function.

```
{zed, state1} = rndKMn(3,3,-1); /* defines the input argument */
zsi = sqrtinv(zed); /* calls the procedure */
```

The procedure with two returns would be called as:

```
{zed, state1} = rndKMn(3,3,-1); /* defines the input argument */
{ret1, zsi } = sqrtinvA(zed); /* calls the procedure */
```

#### 9.4.2 Example 2

Let us develop a procedure that estimates a linear regression model and returns the OLS estimates, their standard errors and t-values. 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;
```

Note: rows, cols, sqrt and diag are global variables known by Gauss.

## 10 GAUSS Functions and keywords

**Functions** are one-line procedures which return a single parameter. They are defined slightly differently:

```
FN fnName(inParam1,... inParamN) = someCode;
```

but otherwise operate in much the same way as procedures. However, the code in a function can only be one line, and functions do not have local variables. Thus functions can be neater than procedures for defining simple repetitive tasks, but apart from that they offer no real benefits. **Keywords** take a single string as input and do not return any output. They can be useful for printing messages to the screen, for example. They are called slightly differently to procedures and functions, looking more like the `PRINT` function. They do allow for local variables and more than one line of code, so in that sense they are more flexible than functions. However, only taking a string as input restricts their value somewhat.

```
keyword what(str);  
    print "The argument is:" str ;  
endp;  
what GAUSS;
```

## 11 Monte Carlo Simulations

Monte Carlo experiments are powerful techniques very often used in applied econometric analyses to assess 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 you 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). Engle and Granger (1987) and Phillips (1996) show 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  $R^2$  are high because of the common stochastic trends.

For the data generating process (DGP), let us generate the following independent bivariate process:

$$\begin{aligned}y_t &= 0.3 + \rho_1 y_{t-1} + \varepsilon_{1t} \\x_t &= 0.5 + \rho_1 x_{t-1} + \varepsilon_{2t}\end{aligned}$$

with

$$\begin{pmatrix} \varepsilon_{1t} \\ \varepsilon_{2t} \end{pmatrix} \sim NIID \left[ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right]$$

We will estimate the static relationship

$$y_t = \alpha + \beta x_t + u_t$$

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  $R^2$ . 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 written by Alain Hecq, results are obtained by considering explicitly the OLS estimator within the loop. Alternatively and more efficiently, the OLS procedure written in one of the examples above can be used. We must keep a trace of the interesting statistics for the computation outside the loop. That is 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;
end;
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,.]);
"";

```

With two independent random walks we should find a significant relationship at a significance level of 5% in 99.72%. In the previous example I have generated independent processes.

## 12 References:

### 12.1 References for this handout

Reorganized and abridged from:

1. "*Introduction to GAUSS Programming Language*" by Eduardo Rossi  
([http://economia.unipv.it/pagp/pagine\\_personal/erossi/gaussnotescomplete.pdf](http://economia.unipv.it/pagp/pagine_personal/erossi/gaussnotescomplete.pdf))
2. "*Guide to Programming in Gauss*" by Felix Ritchie  
(<http://www.trigconsulting.co.uk/gauss/manual.html>)
3. "*Introduction to Gauss - 4034T*" by Alain Hecq  
(<http://www.personeel.unimaas.nl/a.hecq>)
4. "*Command Summary*" by Eric Zivot  
(<http://faculty.washington.edu/ezivot/cmdsum.htm#descriptive>)

### 12.2 Useful GAUSS Resources:

Gauss homepage: <http://www.aptech.com>

Manuals: <http://www.aptech.com/trainingmaterials.html>

[http://www.aptech.com/AS\\_resLibMF.html](http://www.aptech.com/AS_resLibMF.html)

### 12.3 GAUSS Resources for Economists:

*The Econometrics Journal on-line GAUSS list*

<http://www.feweb.vu.nl/econometriclinks/software.html#GAUSS>

*University of Maryland - Marc Nerlove's tutorial "Gauss Programming for Econometricians"*

<http://www.arec.umd.edu/gauss/>

*Florida International University - GAUSS code archives listed by universities and the industry*

<http://www.fiu.edu/~hilljona/gauss%20information.htm>

*American University GAUSS archive - contains e.g. GMM, regression, time series codes*

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

*Bruce Hansen's code (mostly time series)*

[http://www.ssc.wisc.edu/~bhansen/progs/progs\\_subject.htm](http://www.ssc.wisc.edu/~bhansen/progs/progs_subject.htm)

*Eric Zivot's GAUSS resources*

<http://faculty.washington.edu/ezivot/gaussfaq.htm>

*Graphics section of the above*

<http://faculty.washington.edu/ezivot/pqg.htm>

... *AND The One and Only*

[www.google.com](http://www.google.com)

## 13 Appendix: GAUSS Functions and Routines - Quick Reference

This material serves as an overview of basic GAUSS functions and procedures for easier orientation in the numerous GAUSS features. The Help *Reference* section provides a complete description of syntax as well as input / output arguments.

### 13.1 Linear Regression

OLS	Computes least squares regression of data set.
OLSQR	Computes OLS coefficients using QR decomposition.
OLSQR2	Computes OLS coefficients, residuals and predicted values using QR decomposition.

### 13.2 Descriptive Statistics

CROSSPRD	Computes cross product.
MOMENT	Computes moment matrix ( $x'x$ ) with special handling of missing values.
MOMENTD	Computes moment matrix from data set.
DSTAT	Computes descriptive statistics of a data matrix.
CONV	Computes convolution of two vectors.
CORRM	Computes correlation matrix of a moment matrix.
CORRVC	Computes correlation matrix from a variance-covariance matrix.
CORRX	Computes correlation matrix.
MEANC	Computes mean value of every column of a matrix.
STDC	Computes standard deviation of every column.
VCM	Computes a variance-covariance matrix from a moment matrix.
VCX	Computes a variance-covariance matrix from a data matrix.

### 13.3 Cumulative Distribution Functions

CDFN	Computes integral of normal distribution: lower tail.
CDFNC	Computes complement (1-cdf) of normal distribution.
CDFNI	Computes inverse of cdf of normal distribution.
CDFTC	Computes complement of cdf of t-distribution.
CDFTCI	Computes inverse of complement of t-distribution cdf.
CDFTNC	Computes integral of noncentral t-distribution.

### 13.4 Differentiation and Integration Routines

GRADP	Computes first derivative of a function.
HESSP	Computes second derivative of a function.
INTQUAD1	Integrates a 1-dimensional function.
INTSIMP	Integrates by Simpson's method.
INTGRAT2	Integrates over a region defined by functions of x.
INTGRAT3	Integrates over a region defined by functions of x and y.

## 13.5 Root Finding, Polynomial Multiplication and Interpolation

POLYCHAR	Computes characteristic polynomial of a square matrix.
POLYEVAL	Evaluates polynomial with given coefficients.
POLYINT	Calculates Nth order polynomial interpolation given known point pairs.
POLYMULT	Multiplies two polynomials together.
POLYMAKE	Computes polynomial coefficients from roots.
POLYMAT	Returns sequence powers of a matrix.
POLYROOT	Computes roots of polynomial from coefficients.

## 13.6 Random Number Generators and Seeds

Computer programs generate random numbers using a deterministic nonlinear function. There is a key value called *seed* to every such function. Sometimes it is useful to have the same set of random numbers created. This is driven by the *seed value* of the random number generator. In GAUSS the seed is driven by the clock time at which the command is run. Using the same seed, we can repeat generation of the same sequence of random numbers. We can fix the seed using RNDSEED and generate the random numbers using either RNDUS for uniformly distributed numbers or RNDNS for normally distributed numbers. For example:

```
seed = 44435667;  
x = rndus(1,1,seed);
```

### 13.6.1 Random Number Generators

RNDN	Creates a matrix of normally distributed random numbers.
RNDU	Creates a matrix of uniformly distributed random numbers.

### 13.6.2 Random Number Generator Control

RNDCON	Changes constant of random number generator.
RNDMOD	Changes modulus of random number generator.
RNDMULT	Changes multiplier of random number generator.
RNDNS	Creates a matrix of normally distributed random numbers using a specified seed.
RNDSEED	Changes seed of random number generator.
RNDUS	Creates a matrix of uniformly distributed random numbers using a specified seed.