Gauss for Econometrics: An Introduction

R.G. Pierse

1. About *GAUSS*

1.1. What is *GAUSS*?

GAUSS is a matrix-orientated programming language. This distinguishes it both from menu-driven packages such as *MicroFit* and *Pc-Give*, and also from command-driven packages such as *TSP*, *RATS* and *E-Views*. In these other programs, commands are available to compute a particular set of econometric estimators and tests. For each method, a pre-defined set of diagnostics and other information is provided. If a desired method is available in one of these packages, and if the package provides all the required diagnostics, then using the appropriate package is easy and convenient. If what you want to do is not readily available however, then things are more difficult. The command-driven packages have some limited facilities for programming of new estimators and tests, but these may not be flexible enough. *GAUSS*, on the other hand, is extremely flexible and has very efficient built-in procedures for linear algebra and statistics. Any econometric estimator or test can be programmed simply within the language.

It is important to emphasise that *GAUSS* is not specifically an *econometrics* package at all. In fact the set of supplied procedures includes only one econometrics procedure: **ols**. Nevertheless, the matrix nature of the *GAUSS* language makes it very easy to define estimators and tests. As a consequence, *GAUSS* has become very popular with econometricians, and the *GAUSS* language has become a standard medium by which econometricians communicate. A lot of *GAUSS* code is available either freely from its authors, or through third party vendors.

In addition, there is a program, *GAUSSX*, written by Jon Breslaw and available as an optional extra from the *GAUSS* distributors, Aptech, which transforms *GAUSS* into a command-driven econometrics package very like *TSP*. See Pierse (1996) for a review.

1.2. Why use *GAUSS*?

GAUSS has a flexibility not attainable in command-driven packages like *TSP* or *RATS*. On the other hand, it does not have pre-programmed estimation methods and tests, so everything has to be written by the user. In general, therefore, it is only worth using *GAUSS* when you need to use techniques not available in the packages. Monte Carlo and bootstrap simulations are techniques where *GAUSS* has proved a particularly useful tool.

Quite often it is possible to find *GAUSS* code written by someone else that does something close to what you want to do. Sending an inquiry to the *Gaussians* e-mail group is a good way to find out if code already exists. (See next section). However, it is sometimes more work first to understand and then to modify someone else's programs than to write one yourself from scratch.

1.3. Information about *GAUSS*

GAUSS is distributed by ApTech Systems, Inc., 23804 SE Kent-Kangley Rd, Maple Valley, WA 98038, USA. e-mail: *info@aptech.co.* Anderson (1992) reviews the program and Rust (1993) compares its facilities with those of *Matlab.* There are surprisingly few books available on *GAUSS.* Judge *et al.* (1989) is the only book in English and is currently out of print. There is also a two-volume book in French and one in German and in Spanish. The best source of information on *GAUSS* is probably the internet. There are two main channels:

1. An electronic mailing list exists through which users of *GAUSS* can communicate with each other, exchanging information and sharing problems and solutions. Note that this mailing list is quite heavily used and subscribers can expect around 2-5 mail messages per day from fellow *GAUSS* users. To join send e-mail to *Majordomo@mundo.eco.utexas.edu* with the following message: *subscribe gaussians <your e-mail address>*.

2. An archive of *GAUSS* source code is available (for non-commercial use only) from a gopher site run by the Economics department of the American University in Washington D.C. Various *GAUSS* procedures may be found there on topics including cointegration, Bayesian analysis, optimisation. The WWW address for this site is:

gopher://gopher.american.edu:70/11/academic.depts/cas/econ/software/gauss/.

1.4. Versions of *GAUSS*

The rest of these notes relate to version 3.x of *GAUSSi* for the *DOS* operating system. Versions of *GAUSS* for the *UNIX* and *LINUX* operating systems are also available. At the time of writing a new version of *GAUSS* for *Windows 95* and *Windows NT* is in beta-test. This has a number of new features including support for sparse matrices and a built-in procedure *QNewton* for numerical optimisation using quasi-Newton methods. However, at present this Windows version is full of bugs and cannot yet be recommended.

2. The *GAUSS* Language

In representing segments of *GAUSS* code in these notes, the following conventions are adopted: all variables are in *italic* typeface and all *GAUSS* reserved words are in **bold** typeface. Roman typeface is used to represent strings and the names of procedures other than *GAUSS* intrinsic functions.

2.1. Basics

1. The *GAUSS* language is *case insensitive*. Thus *Pi*, *pi* and *PI* all represent the same variable, which is a pre-defined *GAUSS* variable having the scalar value $\pi = 3.141592654\cdots$. Variable names must begin with a letter or underscore '_' followed by a combination of letters, numbers, or the underscore character. The maximum length is 8 characters, and reserved *GAUSS* words such as **log** or **sqrt** are not allowed.

2. Statements in *GAUSS* are terminated by a semi-colon ';'. Comments can be included alongside statements. Anything between the characters '/*' and '*/' is treated as a comment. These comments can be nested. Characters between a pair of '@' characters are also treated as comments but cannot be nested.

3. There is no integer type in *GAUSS* as exists in other languages. All variables are stored in double precision and can have both *real* and *imaginary* components. For example x=3; defines the variable x as a *real* variable taking the value 3.0, whereas x=3+4i; defines x to be *complex* with *real* part 3.0 and *imaginary* part 4.0.

4. Functions in *GAUSS* are called using the general notation

$$\{x,y\} = \mathbf{func}(a,b,c);$$

where **func** is the name of the function and a,b, and c are the function arguments, separated by commas. x and y represent the returns from the function. Unlike other languages, functions in *GAUSS* can return more than one argument. When the function has more than one return, then these are separated by commas and surrounded by curly brackets as in the example. In the special case of a single return, the simpler form

$$x = \mathbf{func}(a,b,c);$$

can be used, or, if the return parameters are not required, then the form

$$\mathbf{func}(a,b,c);$$

is allowed.

5. The basic unit in the *GAUSS* language is a matrix. All variables other than strings are matrices, and most functions operate on matrices. For example, if x is the matrix

$$\begin{bmatrix} 4 & 9 & 1 \\ 0 & 36 & 16 \end{bmatrix}$$

then the *GAUSS* expression **sqrt**(x) results in the matrix

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 6 & 4 \end{bmatrix}$$

which is the element-by-element square-root of the original matrix.

2.2. Operations With Matrices

Matrices are the basic type in *GAUSS* so that matrix operations are the heart of the *GAUSS* language.

2.2.1. Defining Matrices

Matrices may be explicitly defined as in the statement

$$x = \{1\ 2\ 3\ 4,5\ 6\ 7\ 8\};$$

which defines x as the 2×4 matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}.$$

Note that the elements are defined row by row, with spaces separating the columns, and commas separating the rows. The curly braces tell *GAUSS* that this is a definition of a variable.

Matrix elements can be referenced using square brackets. For example x[2,3] represents the element in the second row and third column (which is equal to 7).

Three functions define commonly used matrices: $x =$ **zeros**(7,5); defines x as a 7×5 matrix with all elements set equal to the value of zero. Similarly, $y =$ **ones**(4,6); defines y as a 4×6 matrix with all elements set equal to the value of one. Finally, $z =$ **eye**(12); defines z to be a 12×12 identity matrix, with diagonal elements equal to one and off-diagonal elements equal to zero.

Two useful functions define arithmetic and multiplicative sequences. **seqa**(k,i,n) creates an $n \times 1$ column vector with elements k, $k+i$, $k+2i$, $\cdots$, $k+(n-1)i$. Thus **seqa**$(1,1,n)$ defines an increasing trend and **seqa**$(1,-1,n)$ a decreasing trend. Similarly **seqm**(k,i,n) defines the multiplicative sequence k, ik, i^2k, $\cdots$, $i^{n-1}k$, so that **seqm**$(1,2,n)$ defines a sequence of powers of 2.

2.2.2. Matrix Operators

The standard operators '$+ - * /$' operate in *GAUSS* upon matrices. Thus $C = A * B$; defines C as the matrix product of A and B. Note that the matrices A and B have to be *conformable* for this product to be defined. However, *GAUSS* does also define the result of operations between a matrix and a scalar and between a matrix and a column or row vector.

The matrix operation A/B is defined to give the result $(B'B)^{-1}B'A$ which is $B^{-1}A$ if B is square. The *GAUSS* operator '.*.' is the Kronecker product operator. If A is an $m \times n$ matrix and B is a $p \times q$ matrix, then A.*.B is the $mp \times nq$ matrix $A \otimes B$. The transpose of a matrix is indicated by a prime after the variable name as in A'.

2.2.3. Element-by-element operators

Often we need to compute the conventional scalar multiplication or division operations on each element of a matrix. For this purpose, *GAUSS* provides the element-by-element operators .* and ./ . For example if A and B are matrices

$$A = \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 5 & 2 \\ 4 & 6 \end{bmatrix}$$

then

$$A * B = \begin{bmatrix} 22 & 22 \\ 21 & 26 \end{bmatrix} \quad \text{and} \quad A. * B = \begin{bmatrix} 10 & 6 \\ 4 & 24 \end{bmatrix} .$$

Binary operators such as $+$, $-$, and ˆ (raising to a power) are always element-by-element operators.

2.2.4. Submatrices

Submatrices of a matrix can be specified by ranges as in B=A[2:4,3:7]; which defines B to be equal to the submatrix consisting of rows 2-4 and columns 3-7 of the matrix A. Similarly, C=A[2 4 6,1 3 5 7]; defines C to be a submatrix of rows 2, 4, and 6 and columns 1, 3, 5, and 7 of A. A[2 4 6,·] denotes that *all* the columns are selected.

There is also a function **submat**(A,r,c) which selects the rows and columns of A given by the elements of the vectors r and c respectively. If $r = 0$ ($c = 0$) then all the rows (columns) will be selected.

2.2.5. Concatenating Matrices

There are two operators to join matrices together: '˜' for horizontal concatenation and '|' for vertical concatenation. For horizontal concatenation, the two matrices must have the same number of rows; for vertical concatenation the same number of columns. Complicated matrices can easily be built up in this way, for instance the command:

A = (**eye**(n)˜**ones**(n,1))|(**zeros**(1,n)˜1);

results in the matrix

$$A = \begin{bmatrix} \mathbf{I}_n & \boldsymbol{\iota} \\ \mathbf{0}' & 1 \end{bmatrix} .$$

2.2.6. Matrix Functions

There are a number of intrinsic *GAUSS* functions that perform basic operations of linear algebra:

1. Inversion. The function **inv**(A) returns the inverse of a general square matrix A. If A is symmetric and positive-definite, then the function **invpd**(A) which uses the Cholesky decomposition is more efficient. The generalised Moore-Penrose inverse of a non-singular matrix can be obtained using the function **invswp**(A).

2. Determinants. After using any of the inversion functions, the determinant of the matrix is available in the *GAUSS* variable *detl.* There is also a function **det**(A) to compute the determinant of a matrix.

3. Eigenvalues and vectors. The function $\{va,ve\}$ = **eigv**(A) returns the vector of eigenvalues *va* and the matrix of eigenvectors *ve*, from a general square matrix A. If A is real and symmetric (or complex hermitian) then $\{va,ve\}$=**eighv**(A); is more efficient.

4. Decompositions. The function **chol**(A) returns the Cholesky (upper-triangular) decomposition of a symmetric positive-definite matrix. The function $\{L,U\}$ = **lu**(A) computes the LU decomposition of a general square matrix, **qr**(A) the QR decomposition of a general matrix., and $\{U,S,V\}$ = **svd1**(A) the *singular value decomposition.*

5. Vectorisation. **vec**(A) vectorises a matrix by column. A vector can be reformed into a matrix using the function **reshape**(x,m,n) which creates an $m \times n$ matrix from the elements of the vector x. The matrix is reformed row by row so that the two functions have the relation A = **reshape**(**vec**(A'),m,n).

6. Miscellaneous. **rows**(A) returns the number of rows and **cols**(A) the number of columns in a matrix A. **sumc**(A) returns a column vector, with elements equal to the column sums of A, and **prodc**(A) returns the column products. **maxc**(A) and **minc**(A) return the maximum and minimum elements respectively, in each column of A.

2.3. Conditional Statements and Looping

Like all other high-level languages, *GAUSS* provides facilities for the conditional estimation of statements.

2.3.1. if $\cdots$ endif statements

The sequence of GAUSS commands

```
if x > 3;
   y = zeros(20,1);   /* Command set 1 */
   z = seqa(1,1,20);
elseif x eq 2;
   y = ones(20,1);   /* Command set 2 */
   z = seqa(1,2,20);
else;
   y = 2*ones(20,1);   /* Command set 3 */
   z = seqa(1,3,20);
endif;
```

executes one of three sets of commands according to the value of the variable x. If $x > 3$ then the first set of commands is executed. Otherwise, if $x = 2$, then the second set of commands is executed. Finally, if neither condition is satisfied, then the third set of commands is executed. Each **if** statement must be matched by a corresponding **endif** to mark the end of the conditional block.

The six relational conditions can be specified in either of the two alternative ways as indicated in the table:

$=$	$\neq$	$>$	$<$	$\geq$	$\leq$
eq	**ne**	**gt**	**lt**	**ge**	**le**
==	/=	>	<	>=	<=

When testing conditions on matrices, a condition is only true if it is true for *all* elements of the matrix.

2.3.2. do while loops

Commands can be executed repeatly using a **do while** loop. The sequence of commands

```
i=1;

do while i le 10;

   x[i]=0;

   i=i+1;

endo;
```

executes the commands $x[i]$=0; and i=i+1; 10 times for values of i from 1 to 10. The loop must be terminated by an **endo** statement. Note that the loop index i has to be explicitly updated within the loop, unlike in other programming languages such as Fortran. The **break** command allows premature termination of a loop and the **continue** command moves to the top of the loop to retest the condition.

Note that the **do** loop in the example is equivalent to the single *GAUSS* command x[1:10]=0; and is far less efficient. Wherever possible, loops are to be avoided in *GAUSS* because they are slow to execute.

2.4. Input and Output

2.4.1. File input

In order to use *GAUSS* for econometric analysis, you need to be able to read and write data files. This is perhaps one of the areas where *GAUSS* is weakest. Although it supports its own binary formats for data, .DAT and .FMT, it does not read other binary formats. Hence the most usual form of data in *GAUSS* is the *ASCII* (plain text) format. Text files containing numbers only can be read using the **load** command. For example the command

load d[100,3] = c:\data\mydata.asc ;

reads 100 observations on three variables from the file 'mydata.asc' in the directory 'c:\data' into the 100×3 matrix d . The matrix is filled row by row which is consistent with the data being organised observation by observation. If the data in the file 'mydata.asc' were ordered variable by variable, then the following load statement should be used:

load d[3,100] = c:\data\mydata.asc ; $d = d'$;

2.4.2. Console input

GAUSS programs can request interactive input from the keyboard. The command x = **con**(m,n); requests input from the user of $m \times n$ numbers which are then put in the matrix x. The **wait**; command pauses a program until a key is pressed. This is a useful way to allow results to be inspected as the program is executing.

2.4.3. Output

The *GAUSS* output command is **print**. Since this is the default *GAUSS* command the keyword **print** can actually be omitted. For example the two commands

print "Matrix X: " X; and "Matrix X: "X;

both result in the printing of the string "Matrix X: " followed by the contents of the matrix X. The format of the printing will be determined by *GAUSS*. To control the number of decimal places printed, use the **format** command.

By default output is written to the screen but not saved to a file. In order to save output to a file, an output file needs to be opened before any print statements are executed. The command

output file = c:\out\myoutput **reset**;

opens a file called 'myoutput' on the directory 'c:\out'. The **reset** parameter tells *GAUSS* that if any file with this name already exists it will be overwritten. If the parameter **on** is used in place of **reset**, then output will be appended to the existing file. Output can then be switched off and on again by subsequent **output off**; and **output on**; commands.

Output to the screen can be switched off and on with the **screen off**; and **screen on**; commands.

2.5. Procedures

Procedures are subroutines or functions that take a set of arguments and return a set of values. The intrinsic *GAUSS* commands are all written as procedures. *GAUSS* users can also write their own procedures, and while it is possible to write programs in *GAUSS* without ever making use of procedures, they are a useful way of organising programs into self-contained sections that can subsequently be reused in other programs. *GAUSS* procedures can be recursive so that they can call themselves.

The structure of a procedure is best illustrated by a simple example of a procedure **olsq** to compute OLS estimates.

proc (4) = olsq(y,X);

local n,k,b,seb,e,$s2$,V;

n=**rows**(X); k=**cols**(X);

```
V=invpd(X'X);
b=V*X'y;   /* b is OLS coefficient vector */
e=y-X*b;   /* e is OLS residual vector */
s2=e'e/(n-k);   /* s2 is equation variance σ̂^2 */
V=s2*V;   /* V is variance-covariance matrix */
seb=sqrt(diag(V));   /* seb is vector of coefficient standard errors */
retp(b,seb,sqrt(s2),V);
endp;
```

The procedure takes two arguments: the vector of observations on the dependent variable y, and the matrix of regressors X, and it returns four values: the vector of parameter values b, the vector of parameter standard errors *seb*, the equation standard error **sqrt**(*s2*), and the estimated parameter variance-covariance matrix V.

The first line is the procedure definition. The 4 in brackets after the **proc** indicates that the procedure returns 4 arguments. The final line is the **endp** statement indicating the end of the procedure. The second line declares the 7 variables used within the procedure to be local. Local variables are only known by the procedure being defined. Any variables used within the procedure but not declared within a local statement are global, and will be accessible to all procedures. The function arguments y and X are automatically local and should not appear in a local statement.

The third line uses the **rows** and **cols** functions to determine the number of observations and the number of regressors in the matrices passed as arguments. The following six lines do all the work of the procedure. The penultimate line returns from the procedure with the four return arguments. There can be more than one **retp** statement in a procedure but only one **endp** statement.

The olsq procedure can be called from the main program or another procedure using the line

```
{beta,stderr,s,Var} = olsq(y,X);
```

2.6. *GAUSS* Graphics

GAUSS has some quite powerful graphics routines including the 3-D routines **xyz** and **surface**. Unfortunately, the power comes at some cost in terms of complexity so that these routines can be difficult to use with many options to be set. The most useful graph routine for econometricians is probably the line graph routine **xy** which plots a set of y variables against a single x variable. The following example shows actual and fitted values from a quarterly regression plotted against time.

> **library pgraph**;
>
> **graphset;**
>
> **_plegctl**=1; **_plegstr** = "Actual\000Fitted"; **_pframe** = 0;
>
> **title**("Actual and Fitted Values");
>
> **xy**(**seqa**(1970,0.25,*n*) , *y*~*yfit*);

The first line declares the **pgraph** library. This is necessary in order to be able to use the *GAUSS* graphics routines. The second line resets all the graphics global variables to their default values. These are the variables beginning with the characters '**_p**' that control the options in the graphics routines. It is sensible to reset these to default values before overriding some of the defaults as in the following line. Line 3 sets the global values to select a legend for the graph, and declares the legend text to be 'Actual' for the first y variable in the graph and 'Fitted' for the second. Note that the texts must be separated by the null character '\000'. The command **_pframe** = 0; switches off the box that by default *GAUSS* draws around a graph. The fourth line defines a title for the graph. This will be displayed above the graph. Finally, the **xy** command draws the graph itself. This takes two arguments: the variable to be plotted along the x axis, and the y variable(s) to be plotted against x. In this case, time is represented on the x axis, as defined by the additive sequence **seqa**(1970,0.25,*n*). This will plot quarterly dates along the axis. The second argument is the matrix of variables to be plotted. In this case the matrix is formed by concatenating the two vector variables y and *yfit*.

3. The *GAUSS* Editor

There are two modes of operation of *GAUSS* commands: an interactive mode where commands are executed as they are entered, and a batch mode in which files of commands are built and edited before being executed. The interactive mode is really only useful for very short programs which will only be executed once. The most usual mode of operation is therefore the batch mode, which involves using the *GAUSS* editor. This is a full-screen editor with all the usual facilities. To enter the editor with the file 'myprog.g', type the following command at the *GAUSS* prompt '>':

```
edit myprog.g
```

If the file 'myprog.g' does not exist, then it will be created. When editing is completed, the editor can be exited by the 'CTRL-X' key. This brings up a menu of options: save the file, quit without saving, save and then execute the file etc.

If a *GAUSS* program refers to a procedure called 'procname' that is not included in the main program file being executed, then *GAUSS* will search for a file named 'procname.g', first in the current directory and then in the directories specified in the *SRC_PATH* global. Any file so named will be found automatically, so it is makes sense to use this naming convention when saving procedures. However, it is also possible to create libraries of procedures, with an associated '*.LCG*' file that tells *GAUSS* where to find the source code for each procedure.

References

[1] Anderson, R.G. (1992), 'The Gauss Programming System: a Review', *Journal of Applied Econometrics*, 7, 215-219.

[2] Judge, G.G, R.C. Hill,.W.E. Griffiths, H. Lütkepohl and T-C. Lee (1989), *Learning Econometrics Using Gauss: A Computer Handbook to Accompany Introduction to the Theory and Practice of Econometrics*, John Wiley, New York.

[3] Pierse, R.G. (1996), 'GaussX: Version 3.4', *Journal of Applied Econometrics*, 11, *forthcoming.*

[4] Rust, J. (1993), 'Gauss and Matlab: A Comparison', *Journal of Applied Econometrics*, 8, 307-324.