

# Introduction to GAUSS (II): Data Simulation

Enrique Moral-Benito\*

OCTOBER 2008

## 1 Writing Programs

A program is a structured set of statements that are run together.

To start a new program in GAUSS go to *File - New*, or click the new icon in the *Toolbar*.

As a first command of a GAUSS program, the user can start to give `new`; which clears the workspace. All matrices and procedures (see below) from previous programs are deleted from memory.

Every command within a program must end with a semicolon in order to separate it from the next command (or statement).

You can add comments (a piece of text that is not executed by GAUSS) into the program by typing either `/*this is a comment*/` or `@this is a comment@`

Any active file that contains a program is executed by clicking *Run Active File* in the Run menu or the *Run Active File* icon in the *Toolbar*.

## 2 Flow of Control

Up to now, we have seen how to write programs in a step-by-step way:

```
instruction1;  
instruction2;  
instruction3;  
...
```

Both the loop and the conditional branch involve changes in the flow of control of the program: the sequence of instructions that the program executes, and the order in which they are executed, is being controlled by other instructions in the program. There are two other ways in which the sequence of instructions can be altered: by the suspension (temporary or permanent) of execution; and by procedure calls (see next section).

---

\*E-mail: [enrique.moral@gmail.com](mailto:enrique.moral@gmail.com)

## 2.1 Conditional Branching: IF

The syntax of the full IF statement is:

```
if condition1;
do this1;
elseif condition2;
do this2;
else;
do this3;
endif;
```

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

```
if condition1;
do this1;
endif;
```

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

The following example creates a variable called **b** which will take the value 0 if the variable **a** is equal to zero, the value 1 if **a** is positive and the value -1 otherwise:

```
if a==0;
  b=0;
elseif a>0;
  b=1;
else;
  b=-1;
endif;
```

## 2.2 Loop Statements: WHILE/UNTIL and FOR

GAUSS has two types of loops: **FOR** loops and **WHILE/UNTIL** loops. The loop stops repeating itself when some condition is met. **FOR** loops are used when the number of loops is fixed and known in advance; the others are used when the conditions to enter or exit the loop need to be continually re-evaluated.

- **WHILE/UNTIL** loops. These are used when the conditions to enter or exit the loop need to be continually re-evaluated. These two are identical except that **DO WHILE** loops until condition is false, while the **DO UNTIL** loops until condition is true. The operation of the **WHILE** loop is as follows: **1.** test the condition; **2.** if true, carry out the actions in the loop; then return to stage (1) and repeat; **3.** if false, skip the loop actions and continue execution from the first instruction after the loop. The syntax of these statements is as follows:

<pre>do while condition;   do something;; endo;</pre>	<pre>do until condition;   do something; endo;</pre>
---	--

- **FOR** loops are used when the number of loops is fixed and known in advance. The format of the **FOR** loop is:

<pre>for i(start,stop,interval);   do something;; endfor;</pre>
---

The following example constructs the matrix  $A$  by using a **DO WHILE** and a **DO UNTIL** loops:

$$A = \begin{pmatrix} -2 & 3 & 0 & 0 & 0 \\ 0 & -2 & 3 & 0 & 0 \\ 0 & 0 & -2 & 3 & 0 \\ 0 & 0 & 0 & -2 & 3 \end{pmatrix}$$

<pre>A=zeros(4,5); z={-2 3}; i=1; do while i&lt;=4;   A[i,i:i+1]=z;   i=i+1; endo;</pre>	<pre>A=zeros(4,5); z={-2 3}; i=1; do until i&gt;4;   A[i,i:i+1]=z;   i=i+1; endo;</pre>
--	---

We can also construct another matrix  $B$  with a **FOR** statement:

<pre>B=zeros(4,5); for i(1,rows(B),1); for j(1,cols(B),1);   B[i,j]=i*j; endfor; endfor;</pre>	$\Rightarrow B =$	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \end{pmatrix}$
--	-------------------	--

## 3 Procedures and Functions

### 3.1 Procedures

A procedure allows you to define a new function that you create and which can then be used as if it was an intrinsic function<sup>1</sup>. A procedure is a piece of self-contained code performing an operation,

<sup>1</sup>Intrinsic functions are predefined functions in GAUSS such as  $\ln(x)$ ,  $\text{meanc}(x)$  or  $\text{stdc}(x)$ .

and the procedure can be called from the main program. A procedure is convenient when the same operation has to be performed several times. Wrapping the operation in to a procedure saves you from entering the same piece of code over and over again. The procedures are isolated from the rest of the program.

A procedure looks like:

```
proc (number of returns) = NAME(arg list);  
local local variables;  
proc definition goes here  
other procs, functions, globals, etc.  
may be called and referenced  
retp(return args);  
endp;
```

We see that a procedure definition consists of 5 different parts:

1. Procedure declaration (PROC Statement): the format of the PROC statement is as follows:  
PROC (number of returns) = name(arguments); The arguments (inputs) can be numerous and are then separated by commas. These are the names that are used inside the procedure for the arguments that are passed to the procedure when the latter is called.
2. Local variable declaration (LOCAL statement): local variables are only known within the procedure defined<sup>2</sup>. Remark that there is no information about the size or type of the local variable here. All this statement says is that there are variables which will be accessed during this procedure, and that GAUSS should add their names to the list of valid names while this procedure is running.
3. Body of the procedure: the body contains GAUSS statements that are used to perform the task, they may use intrinsic functions, other procedures...
4. Returns from the procedure (RETP statement): returns a number of items. Their number must correspond with the number of returns in the PROC statement. These returns can however be of any type. It is important to know that GAUSS will not check these returns and it will not warn the user if the number of returns is not equal to the number of returns specified in the procedure declaration. GAUSS will only report an error when the procedure is actually called during a program run.

---

<sup>2</sup>Once the procedure is finished a local variable is deleted from the memory and not accessible for the user in the main program. Note that global variables (i.e. variables that are defined in the main program) are accessible within the procedure. However, global variables cannot be defined inside a procedure.

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

We now illustrate the building and the use of procedures with a small example. We present a procedure that takes as input (argument) one data vector,  $\mathbf{x}$ , and it produces five outputs: its mean(`meanx`), its median (`medx`), its standard error (`stdx`), its maximum (`maxx`) and its minimum(`minx`):

```
proc (5) = desstat(x);
local meanx,medx,stdx,maxx,minx;
meanx=meanc(x);
medx=median(x);
stdx=stdc(x);
maxx=maxc(x);
minx=minc(x);
retp(meanx,medx,stdx,maxx,minx);
endp;
```

## 3.2 Functions

Functions can be interpreted as procedures defined in a single line and that, in general, they produce an output which is a single parameter.

For instance, if we need to evaluate several times the following function:

$$f(x, y) = x^2 + y^3 + \frac{\sqrt{x+y}}{2}$$

we will create our own function in GAUSS and we will call it `funxy` by typing:

```
fn funxy(x,y) = (x^2)+(y^3)+(sqrt(x+y)/2);
```

where the input variables are `x` and `y` and the output variable is `funxy`. As mentioned earlier, `sqrt` is an intrinsic function incorporated in GAUSS for calculating square roots.

Once we have defined the function, if we type:

```
a = funxy(1,2);
```

the function will assign to the variable `a`, the value of the function  $f(x, y)$  evaluated at the point  $(x = 1, y = 2)$ .

Now, we all now a little bit about programming in GAUSS. Therefore we will write our first program for solving the only exercise in this lesson.

## What is a Monte Carlo simulation?

Suppose we have a model given by  $f_X(x; \theta)$ , where  $x$  represents the data and  $\theta$  the parameters. This is what you will see in the blackboard during the following months at CEMFI. However, you should keep in mind that for every single model, there also exists a computer counterpart based on the simulation of  $x$  according to  $f_X(\cdot; \theta)$ .

One type of simulation is **Monte Carlo simulation**, which randomly generates values for random variables over and over to simulate a model.

You may interpret a Monte Carlo simulation as  $S$  controled experiments which replicate the real world given by the model, where  $S$  could be as big as you decide.

Suppose we want to simulate data with a particular distribution  $F$ . If  $F$  is the uniform distribution, GAUSS has the command `rndu`. If  $F$  is normal, we also have a command in GAUSS: `rndn` (remember we used this command in the first lesson). However for the first example we want to simulate data from a t-distribution, and `rndt` does not exist in GAUSS... but there is a very useful result for this situation:

### **Result:**

*Let  $X$  follow  $F$ . Then there exists a  $U$  uniform(0, 1) such that  $X = F^{-1}(U)$*

### **Proof:**

*Take  $U = F(X)$*

Therefore if you want to draw from  $X$ , you have to draw from  $U$  and then evaluate the inverse cdf at  $U$ .

This result also holds when using the complement of the cdf ( $\bar{F} = 1 - F$ ) which is also uniform.

## Exercise 1: Drawing Random Samples

1. Start a new program in GAUSS. In the first line add the command `new`;
2. In the next lines, create two scalars, one with the sample size ( $N = 100$ ) and the other one with the number of Monte Carlo simulations ( $S = 100$ )
3. Simulate  $S = 100$  random samples of size  $N = 100$  from a  $N(0, 1)$  population. (*Hint: GAUSS is a matrix language, so minimize the use of loops and maximize the use of matrices*)
4. By using the result stated above (`cdftci` command in GAUSS), simulate  $S = 100$  random samples of size  $N = 100$  from a  $t(\nu)$  population with  $\nu = 1$ . (*Hint: remember that GAUSS is a matrix language, so minimize the use of loops and maximize the use of matrices*)
5. Calculate the sample means of each of the 100 samples for both the normal and the t-student data. (You will obtain two  $S \times 1$  vectors of sample means)
6. Using a non-parametric kernel density estimator, graph the empirical distribution of the sample means. You can create a procedure with the kernel function (see the appendix if you need help on this issue) What happens with the sample mean in the case of the t-student population with  $\nu = 1$ ?
7. Try using the sample median instead of the sample mean.
8. You can do the same experiment for the case of an exponential distribution (see the appendix if you need help on this issue because there is no command in GAUSS for the exponential cdf...)
9. (Optional) You can repeat the exercise but using loops instead of matrices. You will realize that it is slower.

**Questions:** What happens if you change the value of the kernel's bandwidth? and if you change the sample size from  $N = 100$  to  $N = 10000$ ? and if you use  $\nu = 2$  and  $\nu = 50$  instead of  $\nu = 1$ ?

**Note:** Everytime you want to compare different scenarios, (i.e. different bandwidths or different sample sizes) it is good to have exactly the same samples, for this purpose you need to use the same seed for the random number generator in GAUSS, use the command `rndseed`.

# Appendix

## A1. Kernel Density Estimators

Kernel density estimators belong to a class of estimators called non-parametric density estimators. In comparison to parametric estimators where the estimator has a fixed functional form (structure) and the parameters of this function are the only information we need to store, Non-parametric estimators have no fixed structure and depend upon all the data points to reach an estimate.

To understand kernel estimators we first need to understand histograms whose disadvantages provides the motivation for kernel estimators. When we construct a histogram, we need to consider the width of the bins ( equal sub-intervals in which the whole data interval is divided) and the end points of the bins (where each of the bins start). As a result, the problems with histograms are that they are not smooth, depend on the width of the bins and the end points of the bins. We can alleviate these problem by using kernel density estimators.

To remove the dependence on the end points of the bins, kernel estimators centre a kernel function at each data point. And if we use a smooth kernel function for our building block, then we will have a smooth density estimate. This way we have eliminated two of the problems associated with histograms.

More formally, Kernel estimators smooth out the contribution of each observed data point over a local neighbourhood of that data point. The contribution of data point  $x_i$  to the estimate at some point  $x$  depends on how apart  $x_i$  and  $x$  are. The extent of this contribution is dependent upon the shape of the kernel function adopted and the width (bandwidth or smoothing parameter) accorded to it. If we denote the kernel function as  $K$  and its bandwidth by  $h$ , the estimated density at any point  $x$  is:

$$\hat{f}(x) = \frac{1}{Th} \sum_{i=1}^T K\left(\frac{x - x_i}{h}\right)$$

where  $\int K(t)dt = 1$  to ensure that the estimates  $f(x)$  integrates to 1 and where the kernel function  $K$  is usually chosen to be a smooth unimodal function with a peak at 0. Gaussian kernels are the most often used and `pdfn` is a command preprogrammed in GAUSS that you can use for this purpose. Here you have a line of code that computes the kernel function for a vector `x` in the support vector `xi`:

```
kernel = (1/h)*meanc(pdfn((1/h)*(x-xi')));
```

## A2. Generating Exponential Distributed Data

Exponential with parameter  $\lambda$ :

$$\begin{aligned}f_X(x) &= \lambda e^{(-\lambda x)}, \quad x > 0 \\F_X(x) &= (1 - e^{(-\lambda x)}) 1\{x > 0\}\end{aligned}$$

where  $1\{x > 0\}$  is an indicator function that takes the value 1 if the condition  $x > 0$  is satisfied.

$$\begin{aligned}E(X) &= \frac{1}{\lambda} \\VAR(X) &= \frac{1}{\lambda^2}\end{aligned}$$

Suppose we want to simulate data from the exponential distribution  $F$ .

Let:

$$x = F^{-1}(u)$$

Then:

$$u = F(x) = 1 - e^{-\lambda x}$$

So:

$$x = -\frac{\ln(1 - u)}{\lambda}$$

where  $u \in U[0, 1]$ .

Here you have a line of code that generates  $S$  different samples of size  $N$  of exponential data:

```
xdata = -(1/lb)*ln(1-rndu(n,s));
```