# Introduction to GAUSS (I): From the very beginning

Joan Llull*
CEMFI

October 2009

## 1. What is GAUSS?

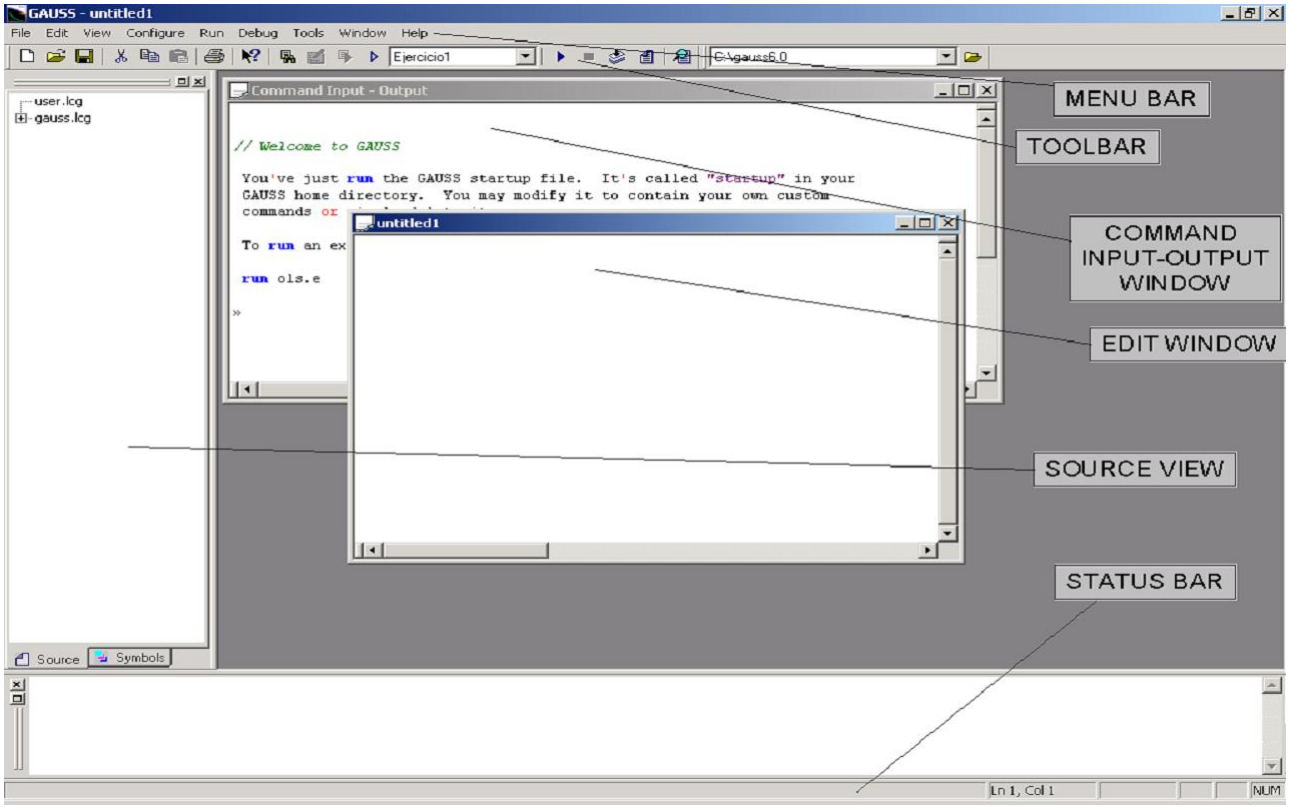
Gauss is a matrix-based programming language that is very popular in the fields of Mathematics, Statistics, and Economics among many others. It is a quite similar language to MATLAB which you may also learn from other courses in this master. Compared to other popular languages such as Fortran or C++, it is usually seen as slower than them but easier to learn and to program, and it has a lot of useful pre-programmed routines that become handy in many applied econometric problems. The fact that it is matrix-based makes it very convenient because it allows to translate the formulas that we want to compute in a very straightforward way. It also provides effective tools for visual analysis of the data and its presentation including a large variety of graphics.

This course is constituted by three lectures. The present lecture aims to provide you with the essentials of GAUSS and its applications. The second one will introduce you to the world of data simulation and, finally, the third one will present you the basics of numerical optimization. The emphasis will be on acquiring familiarity with the fundamentals of GAUSS at the same time that you learn how to approach some problems numerically, but we don't aim to make you an expert!

*E-mail: joan.llull@cemfi.es. URL: http://www.cemfi.es/~joanllull. This notes are based on the class notes used by Enrique Moral-Benito in the academic year 2008-2009; I wish to thank him for providing me with them and for useful comments.

# 2. Workspace

One of the advantages of GAUSS is that it is very user-friendly. It has a Windows based environment, and its main components can be seen in the following screenshot:




From the **Menu Bar** you have access to the usual menus such as File, Edit... which allow you to execute different actions. You can also find there the menu Help in which you can find very complete information about commands, routines, tools, syntax, and many other things.

As it is usual in Windows, some of the previous actions can also be executed from the icons in the **Toolbar**. Moveover, it also allows you to change the working directory and to open program files (see below).

The **Status Bar** shows you the status of the windows and processes on the left side. On the right side you will find the cursor locator and the status of the options OVR, CAP and NUM.

You can work in two different ways with GAUSS. The first one is to write the commands directly on the **Command Input - Output Window** and execute them by clicking the "enter" key in your keyboard. You normally do this for very simple (and probably non-repetitive) exercises. The results of the commands are shown in the same window.

The second (and most common) way of working with GAUSS is by running programs

previously written in any text editor. GAUSS includes its own text editor, the **Edit Window**. If you want to access the GAUSS text editor, there are two different ways:

1. Go to *File-New*
2. Click on the *New* button in the Toolbar

In the new window, you can type the commands you want to execute. Then, when you run the program, the results will be shown in the Command Input - Output Window.

The last (but not least) component of a GAUSS window is the **Source View** that you can see on the left side of the screen. There are two different tabs that allow us to access the files and symbols related to our working space:

- Source Tab: it includes a list of active libraries and their source files. You can access these files by clicking the right button of your mouse.
- Symbols Tab: it includes the symbols of the working space according to their type. (the most used is *matrices*)

# 3. Data Input

There are two basic ways of data input in GAUSS:

1. To create directly vectors and matrices of data in the program
2. To read the data stored in another file

## *3.1. Variable Definition*

### *3.1.1. Scalars*

To create a variable in GAUSS we only have to introduce the name of the variable and its value[1]. For example, the command:

```
a=3
```

creates `a` scalar variable with value equal to 3. Any posterior command can make use of this variable, for example:

```
b=sqrt(a)
```

generates a new variable called `b` whose value is the square root of the variable `a`.

[1] It is important to note that GAUSS is not case-sensitive, i.e., it does not distinguish between upper-case and lower-case letters.

### *3.1.2. Vectors and Matrices*

We can also define vectors and matrices in a similar way than scalars. If we want to create the following matrix:

$$\texttt{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

we type either:

```
A = {1 2,3 4,5 6}
```

or:

```
let A[3,2] = 1 2 3 4 5 6
```

The commands `rows(A)` and `cols(A)` show the number of rows and columns of the matrix `A`.

Standard matrices can be defined as follows:

- Matrix `B` of ones with dimension $N_1$x$N_2$: `B = ones(N1,N2);`
- Matrix `C` of zeros with dimension $N_1$x$N_2$: `C = zeros(N1,N2);`
- Identity matrix `D` with dimension $N_1$x$N_1$: `D = eye(N1);`
- Matrix `F` of threes with dimension $N_1$x$N_2$: `let F[N1,N2] = 3;`

A vector `y` whose elements follow an arithmetic progression can be created as follows:

```
y = seqa(initial value,increment,N)
```

where `N` is the number of elements in the sequence (number of rows). For example:

$$\texttt{y = seqa(1,0.25,4)} \;\Rightarrow\; y = \begin{pmatrix} 1.00 \\ 1.25 \\ 1.50 \\ 1.75 \end{pmatrix}$$

## *3.2. Elements of a matrix*

Given the matrix:

$$\texttt{A}= \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

- To access the element $(i, j)$ of `A` we type `A[i,j]` or `z=A[i,j]` if we want to assign the value of that element to a new scalar variable called `z`.

- To access the row $i$ of the `A` matrix $\Rightarrow$ `A[i,.];` For example: `A[1,.]=` $\begin{pmatrix} 1 & 2 \end{pmatrix}$
- To access the column $j$ of the `A` matrix $\Rightarrow$ `A[.,j];` For example: `A[.,1]=` $\begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix}$
- We can also have access to different sub-matrices, for example:

$$\texttt{A[2:3,1:1]=} \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

## *3.3. Reading Data Files*

There are several alternatives but here we will explain only the most basic way of loading data in GAUSS from an external file. We need to know the number of observations and variables there are in the data file and use the command:

```
load x[r,c]=filename
```

where `x` is the name of the matrix that will contain the data in GAUSS, `r` is the number of rows (observations) and `c` the number of columns (variables). For example:

```
load x[80,44]= matrix1.csv
```
(1)

If we do not know the number of observations but we do know the number of variables, we can load the data as follows:

```
load x[]        = matrix1.csv;
     y  = reshape(x, rows(x)/44, 44);
```

where `m` is the number of variables.

# 4. Basic Operations with Matrices

Given the matrices:

$$A = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \text{ and } B = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

- Multiplication, addition and substraction: `C*A, A+B, A-B` (we should always check that the dimension of the matrices is correct)
- Element by element operation: `F = A.*B` $\Rightarrow F = \begin{pmatrix} 3 \\ 8 \end{pmatrix}$ or `G = A./B` $\Rightarrow G = \begin{pmatrix} 0.33 \\ 0.50 \end{pmatrix}$

- Element by element exponentiation: `^`
- Transpose: `C'`
- Inverse: `inv(C)`
- Horizontal concatenation[2]: `C = A ~B` $\Rightarrow C = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$
- Vertical concatenation: `D = A| B` $\Rightarrow D = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$
- Relational operators: `<`, `>`, `<=`, `>=`, `==`
- Logical operators: `and`, `not`, `or`
- Determinant: `det(C)`
- Elements of the diagonal: `diag(C)`
- Eigenvalues: `eig(C)`

# 5. Graphs

Any program drawing graphs should have the lines:

```
library pgraph;
graphset;
```

ideally at the start of the program. `"library pgraph"` enables GAUSS to know where all the specialized graph-drawing routines are to be found. Note that graphs cannot be drawn if this line is omitted. `"graphset"` resets the graphical global variables to the default state.

Previous to the call to the function that draws the graph, we must define the specifications of the graph: labels, titles and other options. For example:

- To add a title to the graph:

```
title("The title");
```

- To label the axes:

```
xlabel("The x label");
ylabel("The y label");
```

[2] The symbol ~ can be obtained by typing Alt+126

- To define the scale of the axes:

```
xtics(min,max,step,minordiv);
ytics(min,max,step,minordiv);
```

  For example, if we want to define a five-year time scale with year subdivisions for the X-axis between the years 1970 and 2000, we will write:

```
xtics(1970,2000,5,5);
```

- To change the color of the graph lines we modify the value of the global variable `_pcolor`. The color is defined by a integer between 0 and 15. By default, GAUSS sets `_pcolor=0`.

- By default, GAUSS saves all the graphs in the same file, "graphic.tkf". Therefore, if we plot more than one graph that we want to save, we need to assign a name to every file in order to avoid that GAUSS overwrites the files. This is done by typing:

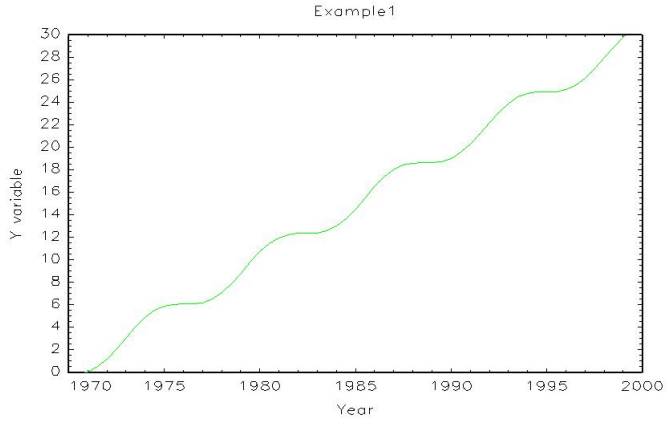
```
_ptek="filename.tkf";
```

After customizing the graph, then we call two dimensions graphs with the following command:

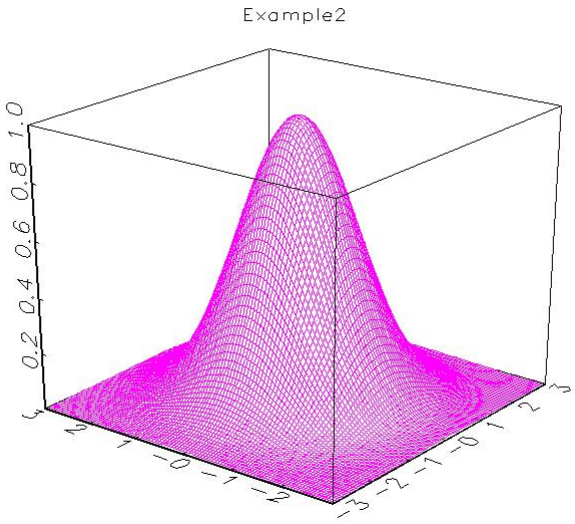
```
xy(x,y);.
```

For example, the plot below graphs the following variables:

```
x=seqa(1970,.5,300);
y=sin(x)+x-1970;
```



You can obtain more information about graphs in GAUSS by typing "Using Publication Quality Graphics" in the *Search* tab of the *User Guide* in the Help Menu. There are other interesting graphs like histograms (you will use them in the exercises below) or nice surface graphs like the following:



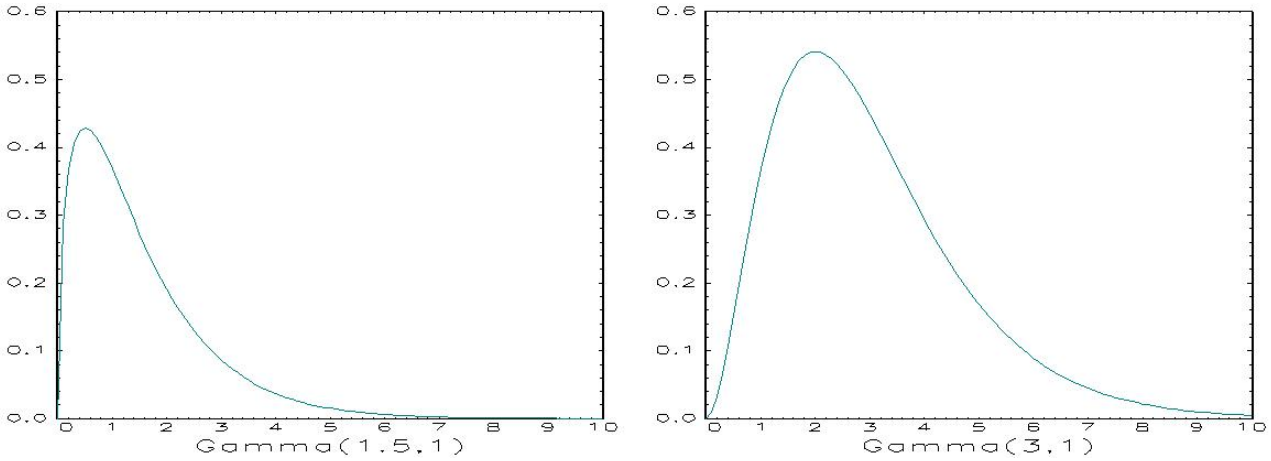
# Code for obtaining the graphs from the Problem Set

**Exercise 1. Gamma distributions**

```
new;
cls;
library pgraph;
graphset;

N=100;

a=0;b=10;

xvar=seqa(a,(b-a)/N,N+1);

alpha=1.5; lambda=1;
yvar1=(xvar.^(alpha-1)).*(exp(-lambda*xvar));

alpha=3; lambda=1;
yvar2=(xvar.^(alpha-1)).*(exp(-lambda*xvar));

_pnum=2;
_pnumht=.25;
_paxht=.35;
_pdate="";
_pcolor=3;
_pmcolor={0,0,0,0,0,0,0,0,15};
_ptek="PS1d.tkf";



xtics(a,b,1,5);
ytics(0,.6,.1,5);
```

```
fonts("simplex simgrma");


window(1,2,0);
setwind(1);
xlabel("\201Gamma(1.5,1)");
xy(xvar,yvar1);
nextwind;
xlabel("\201Gamma(3,1)");
xy(xvar,yvar2);
endwind;
```

The resulting pictures are the following:

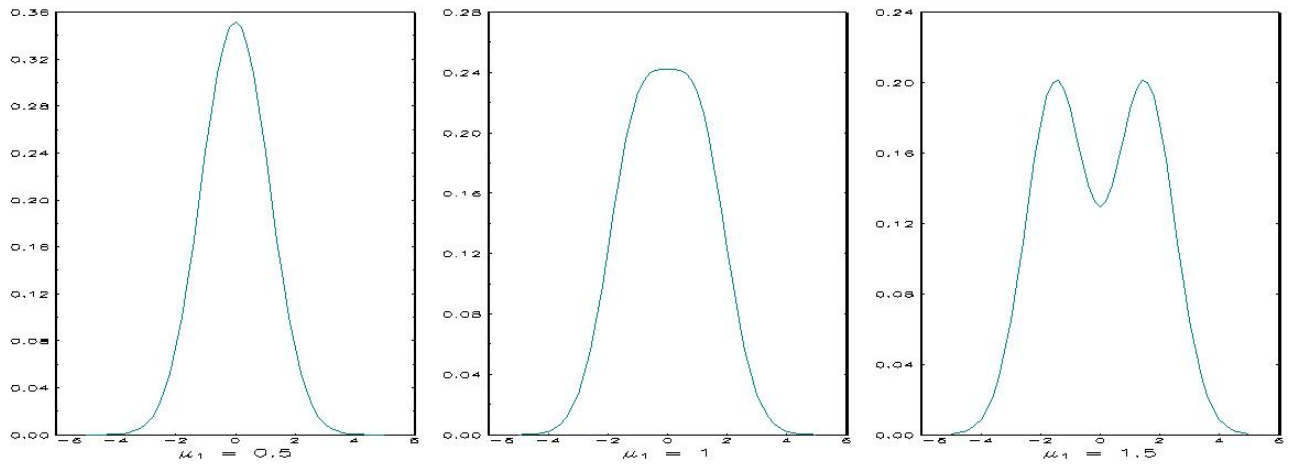



**Exercise 2. Normal mixtures**

```
new; cls; library pgraph; graphset;


N=50;


a=-5;b=5; p1=1/2;p2=1/2; sigma1=1;sigma2=1;


xvar=seqa(a,(b-a)/N,N+1);


mu1=.5;mu2=-mu1;
yvar1=p1/sigma1*pdfn((xvar-mu1)/sigma1)+p2/sigma2*pdfn((xvar-mu2)/sigma2);
```

```
mu1=1;mu2=-mu1;
yvar2=p1/sigma1*pdfn((xvar-mu1)/sigma1)+p2/sigma2*pdfn((xvar-mu2)/sigma2);

mu1=1.5;mu2=-mu1;
yvar3=p1/sigma1*pdfn((xvar-mu1)/sigma1)+p2/sigma2*pdfn((xvar-mu2)/sigma2);

_pnum=2; _pnumht=.25; _paxht=.35; _pdate=""; _pcolor=3;
_pmcolor={0,0,0,0,0,0,0,0,15}; _ptek="PS2d.tkf";

fonts("simplex simgrma");

window(1,3,0); setwind(1); xlabel("\202m\201]1[ = 0.5");
xy(xvar,yvar1); nextwind; xlabel("\202m\201]1[ = 1");
xy(xvar,yvar2); nextwind; xlabel("\202m\201]1[ = 1.5");
xy(xvar,yvar3); endwind;
```

The resulting pictures are the following:



# Exercises

Now we all have some basic notions of the GAUSS programming language. Therefore, we are ready to practise and to do some very easy exercises.

If you have some previous experience in any programming language and you find the exercises too easy, you can help your mates once you have finished the exercises.

## *Exercise 1: Matrices*

Generate the following matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{pmatrix} \quad C = \begin{pmatrix} 2 & 8 & 4 \\ 9 & 3 & 6 \\ 1 & 5 & 2 \end{pmatrix}$$

and compute:

1. $D = B * A$
2. $E = D * A'$
3. $F = B + C$
4. $G = B. * C$
5. Try to compute $A * B$ , $A. * C$ or $A + E$. What happens?
6. Calculate the determinant of $C$ in two different ways and check that you obtain the same result. (*Hint: use the eigenvalues of* $C$)
7. Compute the inverse of $H = A * A'$. What happen? Why?[3]

## *Exercise 2: Random Number Generator*

GAUSS has its own random number generator. This will be very useful for us when we do any kind of simulation. For example, we can generate a vector or a matrix of normal random numbers with the following command:

```
rndn(r,c)
```

where `r` and `c` are the numbers of rows and columns of the matrix of normal random numbers.

[3]Note that $H$ can be written as $A[.,1] * A[.,1]' + A[.,2] * A[.,2]'$.

1. Create a scalar variable called `N` with the value $1,000,000$.

2. Generate a $N \times 1$ vector of normal random numbers and call it `z`.

3. Compute the mean and the standard deviation of `z` and call them `muz` ($\mu_z$) and `stdz` ($\sigma_z$).

4. Using `z`, generate a new vector `z1` with standard deviation 2 and mean 0

5. Generate a $N \times 1$ vector of normal random numbers called `y` with mean to 3 and standard deviation 0.5. ($\mu_y = 3,\ \sigma_y = 0.5$)

6. Check that the mean and the standard deviation of `y` are 3 and 0.5 respectively.

7. Do the exercise again but using $N = 1,000$. What happens?

## *Exercise 3: Graphics*

Now we will plot the vectors of random numbers that we have generated in the previous exercise (the ones with 1,000 observations).

1. Load the `pgraph` library and reset all the graphical global variables to the default values.

2. Create the vector for the X-axes with the command `seqa`.

3. Plot the `z` vector of $N(0,1)$ random numbers and add the title: *Graph 1*

4. Save the graph in your computer with the name "`graph1.tkf`"

5. Plot the three series, `z`, `z1` and `y`, together (*Hint: horizontal concatenation*) in the same graph called *Graph 2*

6. Plot two different random draws with size 1,000 of a $N(0,1)$ variable in the same graph.

7. By using the Help menu plot the histogram of the three series

# Introduction to GAUSS (II): Programming in GAUSS and Data simulation

Joan Llull*
CEMFI

October 2009

## 1. Writing Programs

A program is a structured set of statements that are run together. The main part of the program is called Main Program, and is the portion of the code that executes thread of statements that the user has programmed. This Main Program can make calls to two types of subordinated parts of it which are called *procedures* and *functions* respectively.

To start a new program in GAUSS go to *File - New*, or click the New icon in the *Toolbar*. A program is usually started with the `new;` statement, which clears the workspace; as a result, all matrices and procedures 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*.

*E-mail: joan.llull@cemfi.es. URL: http://www.cemfi.es/~joanllull. This notes are based on the class notes used by Enrique Moral-Benito in the academic year 2008-2009; I wish to thank him for providing me with them and for useful comments.

## 1.1. Procedures

A procedure is a piece of self-contained code, isolated from the main program, that performs an operation used in the main program if it was an intrinsic function[1]. It is very convenient to use them when the same operation has to be performed several times; wrapping the operation into a procedure saves you from entering the same piece of code over and over again.

The structure of a procedure is the following:

```
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 can distinguish 5 different parts:

1. **Procedure declaration** (PROC Statement). This statement includes, apart from the `proc` call itself, its name, its inputs and the shape of its output. It can use more that one input, in which case they should be separated by commas. The shape of the output is defined by the number of elements that are provided as a result of the execution.

2. **Local variable declaration** (LOCAL statement). Local variables, as opposed to global variables, are those variables that are accessible within the procedure only; once the procedure is finished a local variable is deleted from the memory and not longer accessible by the user. Notice that although global variables are accessible within the procedure they cannot be defined inside it. Therefore, this statement specifies the variables that will be accessed during this procedure in order to allow GAUSS to add their names into the list of valid names while this procedure is running. Local variables can take the name of some local variable; in this case, the local variable takes precedence, so the global variable will not be accessible during the execution of the procedure.

3. **Body of the procedure**. The body contains GAUSS statements that are used to perform the task; it may use intrinsic functions, other procedures...

4. **Returns from the procedure** (RETP statement). This statement lists the variables

[1]Intrinsic functions are predefined functions in GAUSS such as ln(x), meanc(x) or stdc(x).

that conform the output of the procedure. Their number must correspond with the number of returns declared in the PROC statement.

5. **End of procedure** (ENDP statement). It tells GAUSS that the definition of the procedure is finished. GAUSS then adds the procedure to its list of symbols, but it does not do any operation until the main program makes a call to it.

We now illustrate the building and the use of procedures with a small example. We present a procedure estimate a linear regression with one covariate and the standard error of the estimates:

```
proc (2) = regression(x,y);
local betavec,xmat,uhat,sigma,stderr,alpha,beta;
xmat=ones(rows(x),1) x;
betavec=inv(xmat'xmat)*xmat'*y;
alpha=betavec[1];
beta=betavec[2];
retp(alpha,beta);
endp;
```

Then, in the main program, we can make calls to the procedure in the following way:

```
{alphahat,betahat}=regression(simx,simy);
```

where `simx` and `simy` are our vectors of data, and `alphahat` and `betahat` are the parameters that we have estimated with the procedure.

## *1.2. Functions*

In GAUSS, functions are single-line procedures which return a single parameter. Therefore, they are convenient in the same situations as in the case of procedures, though they do not define local variables.

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`. Once we have defined the function, if we type:

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

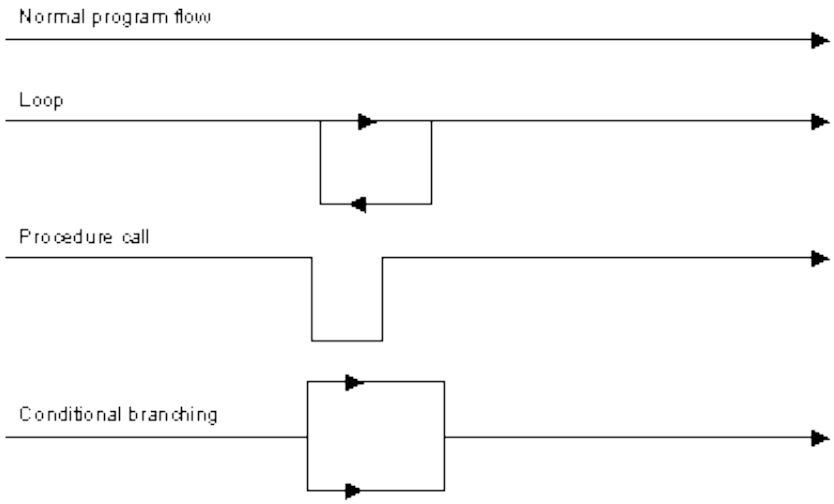
the function will assign to the variable `a`, the value of the function $f(x, y)$ evaluated at the point $(x = 1, y = 2)$.

# 2. Flow of Control

Up to now, we have seen how to write programs in a step-by-step way, i.e, a sequence of commands that are executed one after another. Now we will learn how to alter this sequence to make it more flexible.

There are basically two ways of altering the flow of control of the program: loops and conditional branching[2]. The former repeats some code until some condition is met; the latter chooses which part of the code is executed depending on some conditions.

The following figure shows graphically the main idea beyond each of the alternatives:




## *2.1. Conditional Branching: `IF`*

The syntax of the full `if` statement is:

```
if condition1;
Sequence1;
elseif condition2;
Sequence2;
else;
Sequence3;
endif;
```

[2] Procedures and functions also alter this flow in some way by executing a piece of code that has been written out of the main program.

but all the elseif and else statements are optional.

It is important to note that each condition is tested in the order in which they appear in the program. As a result, the condition2 above is, in fact a double condition: sequence2 is executed if condition1 is not met and condition2 is.

Once the sequence associated with one condition have been executed GAUSS will jump to the end of the conditional branch code and continue execution from there (i.e., GAUSS will only execute one set of actions at most). If none of the conditions is met, then no action is taken, unless there is an else part to the statement, which executes its assigned sequence if none of the previous conditions are satisfied. Therefore, the implicit condition in the else statement above is that neither condition1 nor condition1 is fulfilled.

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

A loop is a sequence of statements which is specified once but which is executed several times in succession. The same sequence will be repeated a fixed number of times (for) or until some condition is met (while/until).

### 2.2.1. FOR loops

These loops are used when the number of iterations of the same sequence is fixed and known in advance. The format of the for loop is:

```
for i(start,stop,interval);
  sequence;
endfor;
```

For example, imagine that we want to calculate the determinant of the matrix

$$C = \begin{pmatrix} 2 & 8 & 4 \\ 9 & 3 & 6 \\ 1 & 5 & 2 \end{pmatrix}$$

by calculating the eigenvalues of the matrix and multiplying them. We can do it with a for loop as follows:

```
c={2 8 4, 9 3 6, 1 5 2};
eigc=eig(c);
determinant=1;
for i(1,3,1);
 determinant=determinant*eigc[i];
endfor;
```

### *2.2.2. WHILE/UNTIL loops*

These loops are often called condition controlled loops because they repeat the sequence of statements until some condition changes (they are, in some sense, a combination of a `for` loop and a conditional `if`). `do while` loops iterate until the specified condition becomes false, while the `do until` loops finish when the condition becomes true.

What these loops do over and over again is to test whether the condition is satisfied or not and then, according to that, either they remain iterating or exit, in which case the main program continues executing the commands placed after the `endo;` statement.

The syntax of these statements is as follows:

```
do while condition;
  sequence;
endo;
```

```
do until condition;
  sequence;
endo;
```

The following example calculates the determinant above by using `do while` and `do until` loops:

```
c={2 8 4, 9 3 6, 1 5 2};
eigc=eig(c);
determinant=1;
i=1;
do while i<=3;
 determinant=determinant*eigc[i];
 i=i+1;
endo;
```

```
c={2 8 4, 9 3 6, 1 5 2};
eigc=eig(c);
determinant=1;
i=1;
do until i>3;
 determinant=determinant*eigc[i];
 i=i+1;
endo;
```

# Exercises

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 a known function $f_X(x;\theta)$, where $x$ represents the data and $\theta$ the parameters. **Monte Carlo simulation** randomly generates values for random variables over and over again to simulate the model. In this case, we simulate $S$ samples of simulated data by using the function $f_X$. Then, for example, we can calculate some statistics from these simulated data and see the properties of these statistics. In sum, you may interpret a Monte Carlo simulation as $S$ controlled experiments which replicate the real world given by the model.

If the data that we want to simulate is gaussian or uniform, then we have preprogrammed commands in GAUSS that make the simulation very straightforward (`rndu` and `rndn`). However, in general, we can consider any pdf $f_X$ which, of course is not preprogrammed in GAUSS. The following result tells us how to deal with this issue:

**Result:**

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

**Proof:**

$U = F(X) = F(F^{-1}(U)) = U$

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

## *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

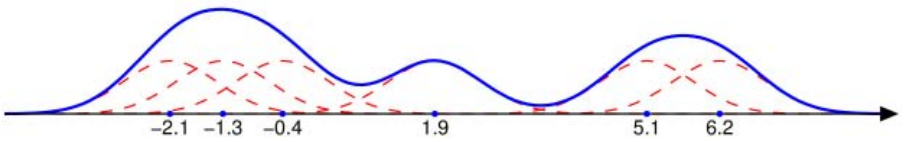
consider the width of the bins (equal sub-intervals in which the whole data interval is divided) and the end points of the bins (where the next bin starts). As a result, the histograms are that they are not smooth, and their shape depend on the width of the bins and on its end points. We can alleviate 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:

$$\widehat{f}(x) = \frac{1}{Nh} \sum_{i=1}^{N} K\left(\frac{x - x_i}{h}\right)$$

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[3]. The following picture provides intuition about how a Gaussian Kernel Estimator does work:




[3] The GAUSS command `pdfn(x)`, which calculates the density of $x$ assuming that it follows a $\mathcal{N}(0,1)$ distribution, can be used for this purpose.

# *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 equal to the sample size ($N = 100$) and the other to 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 (it works faster with matrices than with loops), so minimize the use of loops and maximize the use of matrices.*). **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 of random numbers; for this purpose you need to use the same seed for the random number generator. In GAUSS, place the command `rndseed #;` (where # is a number) before starting to simulate.

4. Following the **result** stated above, simulate $S = 100$ random samples of size $N = 100$ from a $t(\nu)$ population with $\nu = 1$. (*Hint: the GAUSS command* `cdftci(x,v)` *calculates the cumulative density of* $x$ *if it follows a* $t(v)$.)

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. (*Hint: the GAUSS command* `meanc(x)` *generates an* $N \times 1$ *vector with the sample means of each of the columns of the* $N \times M$ *matrix* $x$.)

6. Using a non-parametric kernel density estimator, graph the empirical distribution of the sample means. You may create a procedure with the kernel function. (*Hint: in GAUSS, if you sum or subtract a row and a column vector, the result is a matrix with elements* $(i, j)$ *equal to sum or subtraction of the element* $i$ *of the column vector and the element* $j$ *of the row vector.*) 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.(*Hint: the GAUSS command* `median(x)` *is the analogous to* `meanc(x)` *for the median.*)

8. 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...). You may create a function with the inverse of the exponential cdf.

9. Repeat the exercise 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 $\nu = 1$?

# Appendix: Generating Exponential Distributed Data

Exponential with parameter $\lambda$:

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

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));
```

# GAUSS code for the solution of the exercises in Lecture 1

```
new;
cls;
@-----------------------------------------------------------@
@                       EXERCISE 1                          @
@-----------------------------------------------------------@
A={1 2,3 4,5 6};
B={2 4 6,8 10 12,14 16 18};
C={2 8 4,9 3 6,1 5 2};
D=B*A;
E=D*A';
F=B+C;
G=B.*C;
detc1=det(C);
eigc=eig(C);
detc2=eigc[1]*eigc[2]*eigc[3];
H=A*A';
Hbis=A[.,1]*A[.,1]'+A[.,2]*A[.,2]';
deth=det(H);
rankh=rank(H);
@-----------------------------------------------------------@
@                       EXERCISE 2                          @
@-----------------------------------------------------------@
N=1e4;
z=rndn(N,1);
muz=meanc(z);
stdz=stdc(z);

z1=2*z;

muy=3;
stdy=0.5;
y=muy+stdy*rndn(N,1);
@-----------------------------------------------------------@
@                       EXERCISE 3                          @
@-----------------------------------------------------------@
library pgraph;
graphset;
x=seqa(1,1,N);

_ptek="graph1.tkf";
title("Graph 1");
xy(x,z);

_ptek="graph2.tkf";
title("Graph 2");
xy(x,z1~z~y);

rn1=rndn(1000,1);
rn2=rndn(1000,1);
x1=seqa(1,1,1000);
_ptek="graph3.tkf";
title("Graph 3");
xy(x1,rn1~rn2);

_ptek="hist_z.tkf";
title("Histogram z");
{ breakpoint,midpoint,frequency } = hist(z,30);

_ptek="hist_z1.tkf";
title("Histogram z1");
{ breakpoint,midpoint,frequency } = hist(z1,30);
```

# Introduction to GAUSS (III): Numerical Optimization

Joan Llull*
CEMFI

October 2009

## 1. Numerical Optimization

Probably the tool that is more often used by economists is optimization. In some cases, the functional forms of either the objective function or the first order conditions allow us to solve the problem analytically, but in many other cases they not. In this latter case, we need to know how to solve the problem numerically, and this is the purpose of this third lesson.

Numerical analysis can be understood as a set of evaluations of some particular function that allows us to understand the shape of the evaluated expression. For example, imagine that we are interested in the derivative of a function which we don't know how to differentiate; we can use the definition of the derivative,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

and evaluate it (using a sufficiently small $h$) at many different values of $x$ and then draw a graph with the results. In fact, we have already done a kind of numerical analysis when we performed and drew kernel density estimates last week.

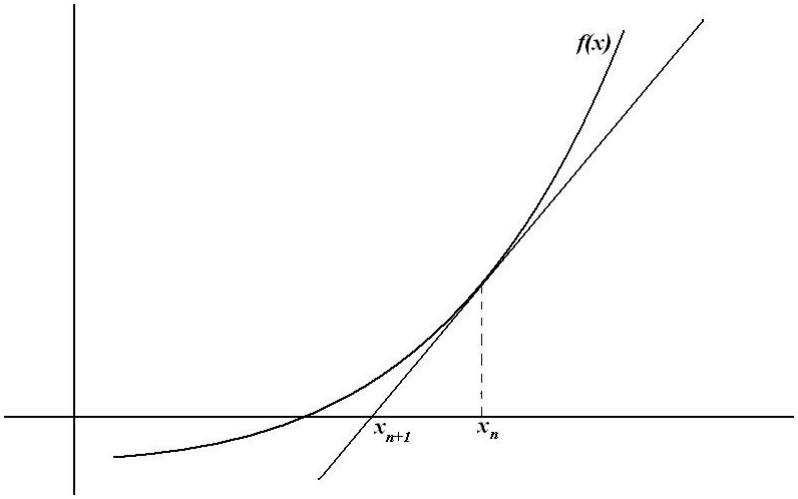
### *1.1. Newton's method (a.k.a Newton-Raphson method)*

The original version of the method is conceived to find the roots of a given function. There are two very important examples in economics in which we are interested finding the zeroes of

*E-mail: joan.llull@cemfi.es. URL: http://www.cemfi.es/~joanllull. This notes are based on the class notes used by Enrique Moral-Benito in the academic year 2008-2009; I wish to thank him for providing me with them and for useful comments.

a real-valued function: equating first order conditions to zero to optimize a problem or finding the fixed point of a function.

The intuition is to approximate the function in a given point, $x_n$ by its tangent line, and then to find the intercept of this line with the x-axis ($x_{n+1}$). The point of doing so is that $x_{n+1}$ will be a better approximation of the root than $x_n$:




Let's see it more formally. Assume that we want to find the root of a function $f(x)$ which is at least differentiable once. We know that the derivative of the function must be equal to the slope of the hypotenuse of the triangle above. Therefore:

$$\frac{f(x_n)-0}{x_n-x_{n+1}} = f'(x_n) \Rightarrow x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1}$$

This expression provides an iterative rule that we can execute starting from a point reasonably close to the true root, $x_0$ and updating it with (1) until reaching convergence, i.e. until $|x_{n+1} - x_n| < \varepsilon$ with $\varepsilon$ arbitrarily small.

## *1.2. Application of Newton's method to optimization problems*

As noted above, we may be interested in minimizing a function rather than obtaining its roots. This may be seen as a special case of the previous expression because minimizing a function is equivalent to finding the roots of its derivative. Therefore, the Newton-Raphson algorithm for minimization is given by[1]:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}. \tag{2}$$

[1] Notice that in this case we need $f(x)$ to be at least twice differentiable.

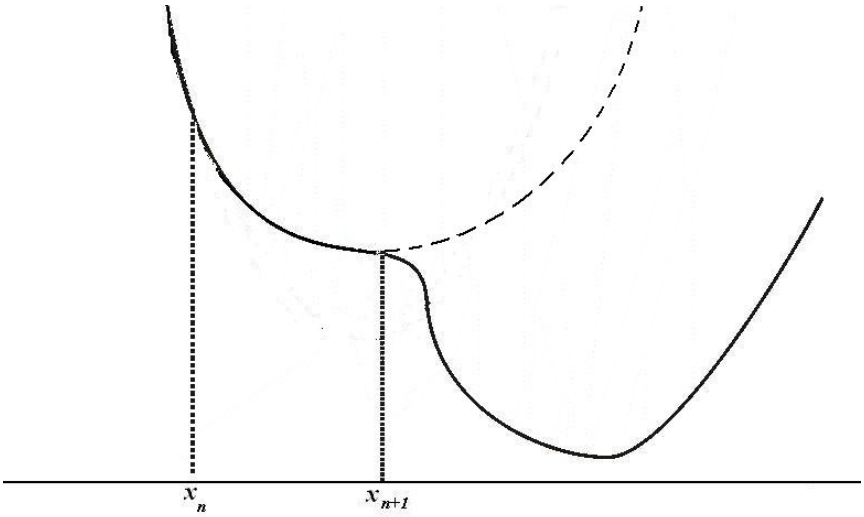
We will see now an alternative interpretation of (2). Consider the following second order Taylor approximation of $f(x)$ around the point $x_n$:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2}(x - x_n)^2.$$

The idea is to update our guess, $x_n$, with the minimum of this approximation:

$$f'(x_n) + f''(x_n)(x - x_n) = 0 \Rightarrow x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)},$$

which provide the updating function from equation(2). Graphically:




## *1.3. The multivariate case*

This method can be used also to multivariate functions. Consider $\mathbf{f} : \mathbb{R}^K \to \mathbb{R}^K$, at least once differentiable. We want to solve

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

where $\mathbf{f}(\mathbf{x})$ and $\mathbf{0}$ are $K \times 1$ vectors and $\mathbf{x} = (x_1, ..., x_K)'$. Then a Newton iteration is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{f}'(\mathbf{x}_n)]^{-1}\mathbf{f}(\mathbf{x}_n) \tag{3}$$

where $\mathbf{f}'(.)$ is the $K \times K$ Jacobian matrix.

For the case of optimization, consider a function $\mathbf{g} : \mathbb{R}^K \to \mathbb{R}$ at least twice differentiable. We want to find the vector $\mathbf{x} = (x_1, ..., x_K)'$ that minimizes $\mathbf{g}(.)$. Now a Newton iteration will be:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [H_g(\mathbf{x}_n)]^{-1}\nabla_g(\mathbf{x}_n), \tag{4}$$

where $H_g(.)$ is the Hessian matrix of $\mathbf{g}(.)$ and $\nabla_g(.)$ is its gradient vector.

An important remark is due here. Imagine that we can calculate analytically the first derivative of the function $f(x)$. Then, we have to decide whether to apply (i) equation (3) on $f'(x)$ (by calculating the Jacobian of $f'(x)$ numerically) or (ii) equation (4) on $f(x)$ (by computing both the gradient and the Hessian numerically). We would always perfer (i). Why? First, because it is much faster (Hessians can be very costly to compute numerically); and second, because it is more precise.

## 2. OPTMUM LIBRARY IN GAUSS

`Optmum` is a GAUSS library that optimizes a function with respect to a set of parameters using Newton's method we have just seen.

As in the case of graphs, to use the `optmum` library you have to explicitly include it in your code by using the command `library optmum;`. The function to be minimized must be provided by you and defined in a procedure.

`Optmum` is called in the same way we called procedures last week:

```
{arg,obj,grad,retcode}=optmum(&func,start);
```

The inputs of the procedure are:

`start`: it is the vector of initial values for the parameters (our $x_0$ above).

`&func` is a pointer to the procedure that computes the function to be minimized. This procedure must have one input argument, a vector $x$, and one output argument, a scalar $f(x)$.

The output consists of the following:

`arg`: the vector $x_{min}$

`obj`: the scalar $f(x_{min})$

`grad`: the gradient $\nabla_f(x_{min})$

`retcode`: scalar, return code. If normal convergence is achieved then `retcode=0`, otherwise a positive integer is returned indicating the reason for the abnormal termination: **1**. forced exit; **2**. maximum iterations exceeded; **3**. function calculation failed; **4**. gradient calculation failed; **5**. Hessian calculation failed...

# Exercises

We will make use of the `optmum` library in order to maximize two different functions.

## *Exercise 1: The Univariate Case*

1. Start a new program (`new; cls;`), create a scalar $N = 1000$ and load the pgraph and optmum libraries.

2. Generate a random sample of size $N$ from a normal distribution with $\mu = 2$ and $\sigma = 1$. (*Hint:* `rndn` *command*)

3. Write a procedure with the function:

$$L(\mu) = -\sum_{i=1}^{N} \frac{(x_i - \mu)^2}{2}$$

   where $\mu$ is the input of the procedure, $L(\mu)$ is the output of the procedure and $x_i$ ($i = 1, ..., N$) is i-th element of the vector that you have generated in step 2.

4. Plot the function. Where is the maximum? (*Hint: create the support vector with the* `seqa` *command and evaluate the function in this support*)

5. By using the optmum library, maximize the function $L(\mu)$ with respect to $\mu$. (*Hint: remember that* `optmum` *minimizes functions, but maximizing* $f(x)$ *is equivalent to minimizing* $-f(x)$)

6. What is the result?

7. Repeat the step 5 but using very different initial guesses (`start`). What happens?

# *Exercise 2: The Multivariate Case*

1. Start a new program (`new;`) in a new file, create a scalar $N = 1000$ and load the optmum library.

2. Generate a random sample of size $N$ from a normal distribution with $\mu = 2$ and $\sigma = 1$. (*Hint:* `rndn` *command*)

3. Write a procedure with the function:

$$L(\mu, \sigma) = -\frac{N}{2}\ln(2\pi) - \frac{N}{2}\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i - \mu)^2$$

   where $\mu$ and $\sigma$ are the inputs of the procedure, $L(\mu, \sigma)$ is the output of the procedure and $x_i$ $(i = 1, ..., N)$ is i-th element of the vector that you have generated in step 2. (*Remark: the input of the procedure is now a 2 by 1 vector*)

4. By using the optmum library, maximize the function $L(\mu, \sigma)$ with respect to $\mu$ and $\sigma$. (*Hint: to create the initial guess, proceed as follows:* `start={5,5};`)

5. What is the result?

6. Repeat the step 4 but using very different initial guesses (`start`). For example try `start={78,95};`. What happens?

# GAUSS code for the solution of the exercises in Lecture 2

```
@--------------------------------@
@--------------------------------@
@ -        EXERCISE 1           -@
@ -  DRAWING RANDOM SAMPLES     -@
@--------------------------------@
@--------------------------------@


@-----------@
@   STEP 1  @
@-----------@
new; cls;
library pgraph;
graphset;
rndseed 23;                          /* seed for the random number generator *


@-----------@
@   STEP 2  @
@-----------@
n=100;                               /* sample size */
s=100;                               /* number of drawn samples */
h=n^(-1/5);                          /* kernel's bandwidth */


@-----------@
@   STEP 3  @
@-----------@
ndata=cdfni(rndu(n,s));              /* normal data     */
/* or we can also do it by: ndata=rndn(n,s);          */


@-----------@
@   STEP 4  @
@-----------@
v=1;                                 /* d.f. of the t distribution */
tdata=cdftci(rndu(n,s),v);           /* t-student data  */


@-----------@
@   STEP 5  @
@-----------@
samplemeansn=meanc(ndata);           /* sample mean with n-data */
samplemeanst=meanc(tdata);           /* sample mean with t-data */


@-----------@
@   STEP 6  @
@-----------@
{tk,sop}=kernel(samplemeanst);
{nk,sop}=kernel(samplemeansn);
xy(sop,tk~nk);                      /* graph of sample mean estimators */


@-----------@
@   STEP 7  @
@-----------@
samplemedianst=median(tdata);          /* sample median with t-data */
{tkm,sop}=kernel(samplemedianst);
/*xy(sop,tk~tkm);*/                        /* graph of sample mean and median


@-----------@
@   STEP 8  @
```

```
@-----------@
lb=4;
xdata=-(1/lb)*ln(1-rndu(n,s));
samplemeansx=meanc(xdata);
{xk,sop}=kernel(samplemeansx);
/*xy(sop,xk);*/


@-----------@
@   STEP 9  @
@-----------@
/* Example: it does the same as steps 3 and 4 but using a loop */
ndata=zeros(n,s);
tdata=zeros(n,s);
v=1;
i=1;
do while i<=s;
    nd1=rndn(n,1);
    td1=cdftci(rndu(n,1),v);
    ndata[.,i]=nd1;
    tdata[.,i]=td1;
    i=i+1;
endo;



/* Procedure with the kernel function */
proc(2)=kernel(x);
local a,b,xi,kernel;
a=-3; b=3;
xi=seqa(a,(b-a)/n,n);
kernel=(1/h)*meanc(pdfn((1/h)*(x-xi')));
retp (kernel,xi);
endp;
```