GAUSS, an introduction[1]

Index

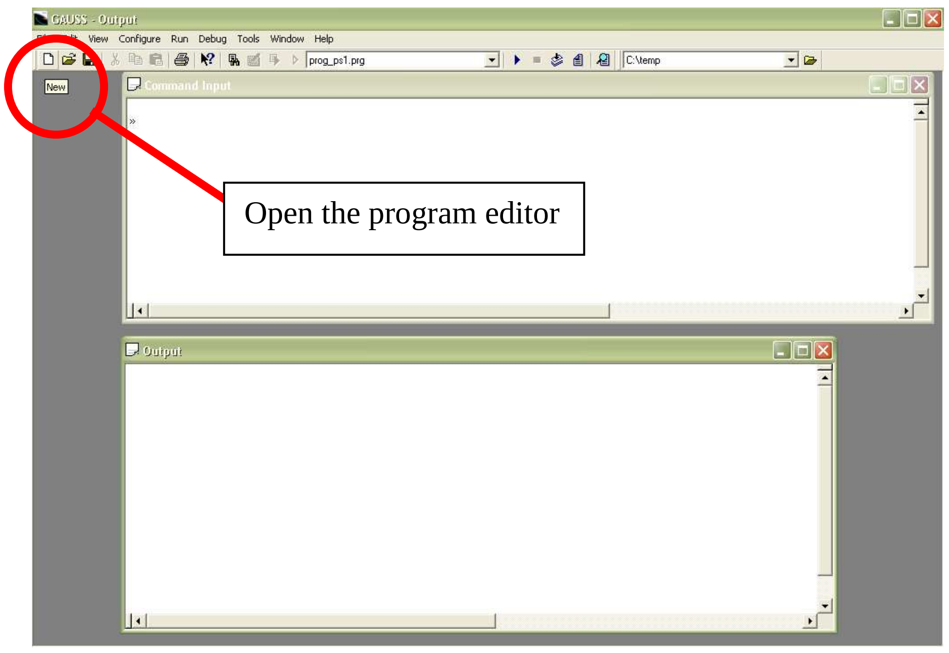
[1] A.j.goujard@lse.ac.uk. Comments and suggestions to improve this first draft are more than welcome. Please do not quote, report or diffuse.

This handout explains how to use GAUSS to program the estimators and matrices manipulations required for the EC475 exercises. Good programming practices and tricks to improve your programs will be provided later on. More advanced tools may be available but this document focuses on the programming structure and “how to do that” rather than on “why doing that” or “how to do that faster”.

1. The software

The first thing to know about GAUSS is that everything is created so as to manipulate matrices, so do not expect to be able to visualize your data as in STATA, SAS… or to have built in OLS routines. Thus more or less everything has to be programmed. However GAUSS is efficient at creating users’ developed programs and fast at running them! Moreover, contrary to pure programming languages, GAUSS as MATLAB can directly manipulate matrices and has a lot of tools to transform them. It also contains pre-programmed features such as numerical optimization routines that are useful for applied econometrics.

1.1 The software screen



The basic GAUSS screen has 2 main windows[2]:

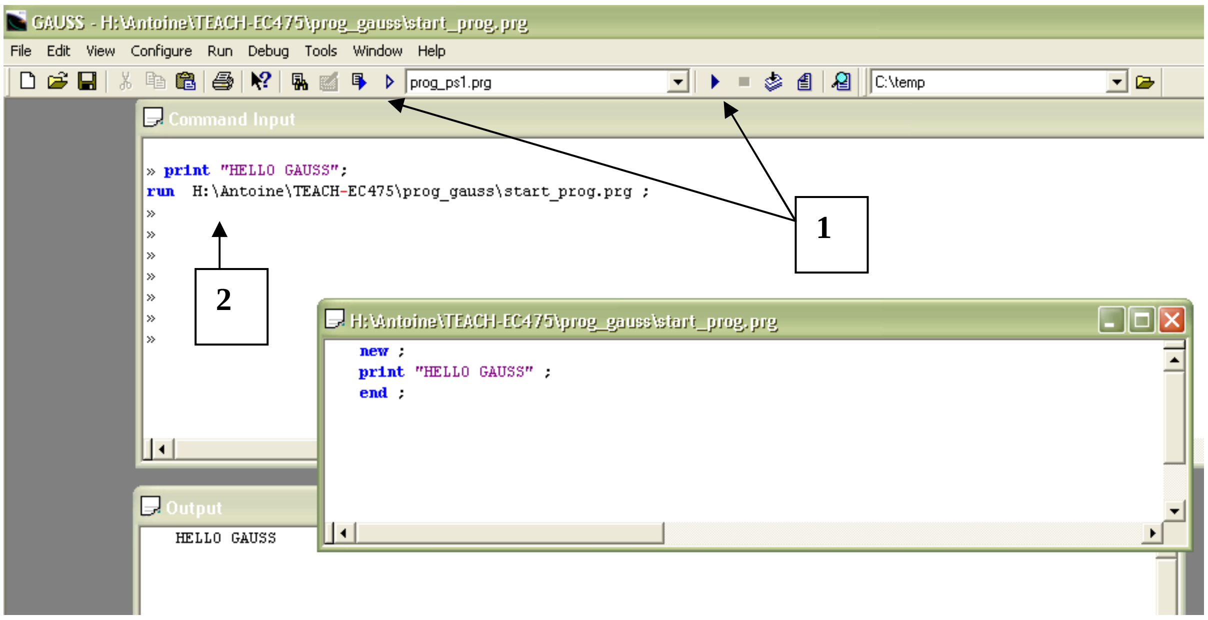
- **<u>command input:</u>** where you can enter your own command one by one ;
- **<u>output:</u>** where you see the results of these commands.

Eg. Typing in the command window: **print "HELLO GAUSS" ;**
Returns: HELLO GAUSS in the output window.

[2] If you do not see them directly, try manipulating the windows’ sizes.

1.2 Writing a program

In this course, we want to be able to store the program and to combine several lines of code. So we have to create a program or code file. GAUSS has an internal program editor that you can open by clicking on the white sheet on the left this opens a third window[3]. You can immediately save your program wherever you want using the "save" icon. A gauss program file has a **.prg extension**. This should look like this:


GAUSS - H:\Antoine\TEACH-EC475\prog_gauss\start_prog.prg
File Edit View Configure Run Debug Tools Window Help
prog_ps1.prg
C:\temp
Command Input
1
2
H:\Antoine\TEACH-EC475\prog_gauss\start_prog.prg
Output

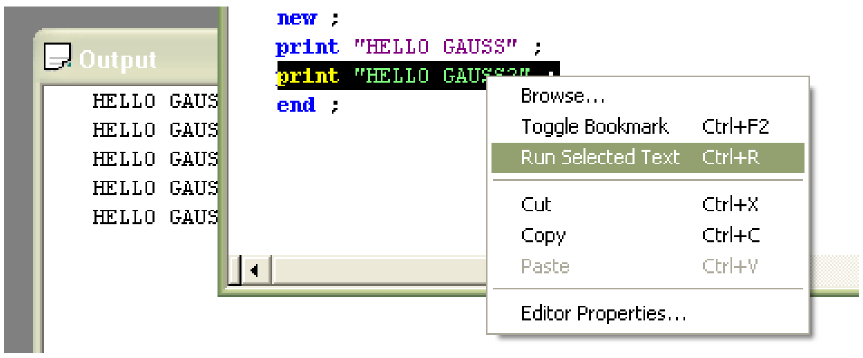
```
» print "HELLO GAUSS";
run  H:\Antoine\TEACH-EC475\prog_gauss\start_prog.prg ;
```

```
new ;
print "HELLO GAUSS" ;
end ;
```

```
HELLO GAUSS
```

Your program should have a clear start "**new ;**" and an end: "**end ;**". You can run your program in (at least) 2 ways:

1/ directly from the program editor using one of the blue arrows on the top of the screen **[1].** This allows you to select only some lines of code (using run selected lines on the top panel or by right clicking on your selected lines):



```
new ;
print "HELLO GAUSS" ;
print "HELLO GAUSS" ;
end ;
```

```
HELLO GAUS
HELLO GAUS
HELLO GAUS
HELLO GAUS
HELLO GAUS
```

2/ from the command window [2]:
run "H:\\Antoine\\TEACH-EC475\\prog_gauss\\start_prog.prg" ; /* always more secure to delimit the path using quotes then need \\ for path */

[3] An other solution is to use any text editor you like, write your program, save it and rename it with a .prg extension.

The output windows may also return you many error codes. For example if I try to run:

print "HELLO GAUSS ; /* bad programming */

I obtain: (0) : error G0097 : String not closed

The first number inside the parenthesis corresponds to the line of your program where the mistake occurs. As you can see **you can and should include comments in your gauss programs**. This can be done using:

1. /* this is a comment */
2. @ this is a comment @
3. /* this is a very long
   Comment */

I prefer the first one which is valid in many other statistical and econometric software. You can type long comments on several lines as in example 3.

The GAUSS program editor also supports many keystroke shortcuts. Among those:

1. CTRL-X delete the selected text
2. CTRL-R execute the selected block of commands

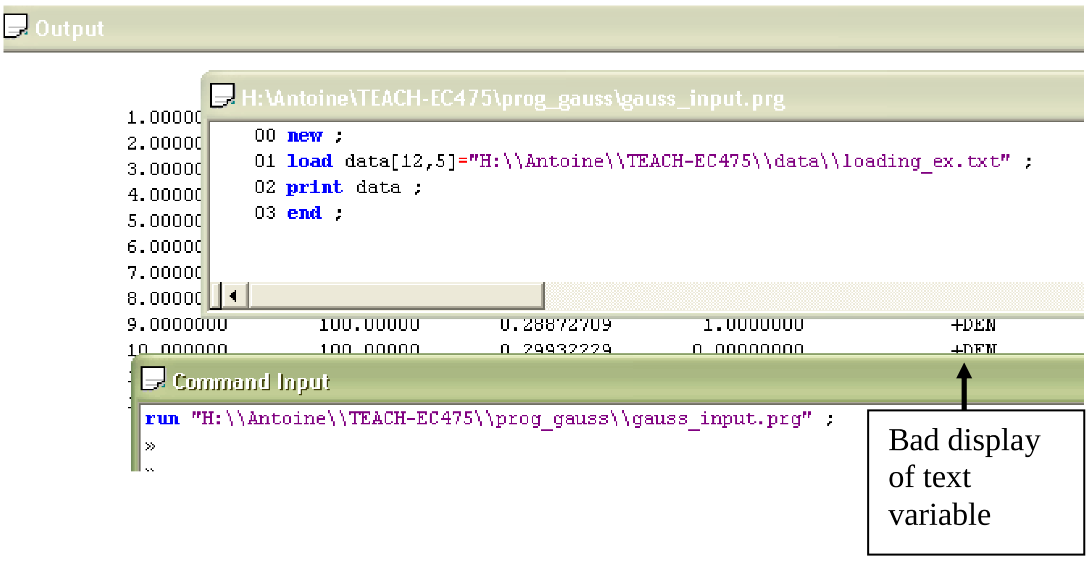
It is also very helpful to use the option that **displays the lines' indices** (right click under editor's properties).

2. Loading data

Once we know how to create a program, the second step for econometrics is to be able to access the data. This depends on the way you have access to them. I detail three cases: text-files, GAUSS files and creating your own matrices for simulation in part 3. I assume that you may convert all other data formats in either a GAUSS or text file. At LSE, this can be done in a straightforward way by STATTRANSFER which you can find on the LSE network under *programs \\ statistics \\ stattransfer9*.

2.1 Loading Ascii/text files

Here I load a data set of 12 observations and 5 variables:


Output
H:\Antoine\TEACH-EC475\prog_gauss\gauss_input.prg
Command Input
Bad display of text variable

```
00 new ;
01 load data[12,5]="H:\\Antoine\\TEACH-EC475\\data\\loading_ex.txt" ;
02 print data ;
03 end ;
```

```
1.00000
2.00000
3.00000
4.00000
5.00000
6.00000
7.00000
8.00000
9.0000000       100.00000      0.28872709      1.0000000         +DEN
10.000000       100.00000      0.29932229      0.00000000        +DEN
```

```
run "H:\\Antoine\\TEACH-EC475\\prog_gauss\\gauss_input.prg" ;
»
»
```

This creates a 12*5 matrix called data (see part 3 on matrix). We could have proceed in two steps which may be useful if we are unsure about the size or quality of our data.

new ;
load data[]=”H:\\Antoine\\TEACH-EC475\\data\\loading_ex.txt” ; /* load the data as a column vector */
print data ; /* display the small dataset loaded in the output */
data1=reshape(data,12,5) ;[4]
print data1 ; /* display the reshaped matrix */
print $data1[.,5] ; /* display properly the last column of names */
end ;

Rk1: **data1[.,5]** selects all the rows of data1 and only column 5.
Rk2: GAUSS truncates text variables to 8 characters. Eg. XXXYYYZZZ becomes XXXYYYZZ.

A last important thing is to be able to assign variables’ names to the elements of your created matrices. This can be done by creating a GAUSS file from the original data:

Let vnames=X1 X2 X3 X4 XNAME ; /* vnames is a variable containing var names */
Outname=”H:\\Antoine\\TEACH-EC475\\data\\ex1” ; /* this creates the name of the new data set ex1 stored at the required path */
Create outfile=^outname with ^vnames,0,8 ; /* this creates and opens the data set with outname=ex1 and variables’ names in vnames, 0 sets the # of columns to correspond to # of var in vnames, 8 corresponds to the highest storage precision available */
Nrows=Writer(outfile,data1) ; /* this writes the created matrix data1 into the specified outfile and stores the number of writted rows into nrows */
Print nrows ;; print “nrows written to the data” ; /* check!! The ;; instead of ; specifies that you want the next thing printed on the same line */

Why is this useful? Now suppose we want to see the last column of our data. A first solution is:
print $data1[.,5] ;

This clearly depends on the way the data are stored and if you are not careful, this can cause a large number of mistakes. So instead of using the column number (5) to index the program, we want to use the variable name (XNAME).
print $data1[.,ixname] ; /* note the i for index before xname */

This will not work! Before we can actually do that we need to open our created GAUSS data set ex1 as explained in the next paragraph.

[4] This assumes we know both the number of observations and number of variables. If we know only that the data has 5 variables, we could use: **data1=reshape(data,nrows(data)/5,5) ;** see the matrix part of the handout for details.

2.2 Loading Gauss files

Gauss data files come in pairs (at least if you are using windows on your computer). The two parts are:

1. .dat the matrix of the data
2. .dht the list of the names of the variables

Loading a data in GAUSS takes also two steps:

1. opening the data
2. reading the data in an appropriate matrix!

Here is an example from PS1:

```
new ;
CLOSEALL ; /* this closes all datasets previously opened */
filename="H:\\Antoine\\TEACH-EC475\\data\\lqdata" ;
/* the double backward slashes are needed to specify the path within the quotes */
/* creates a string macro filename */
/* the name of the storage file is lqdata */
/* gauss stores the data into 2 parts .dat=matrix, .dht=var names */
open input=^filename FOR UPDATE VARINDXI ; /* opens the dataset */
data=READR(input,ROWSF(input)) ; /* reads it into some matrix called data */
/* rowsf(input) specify we want the S observations. */
/* It can be changed for 10, 100 etc. */
vnames=getname(filename) ; /* takes the names of the variables */
end ;
```

Rk1: you may want to have a look at the variables' names which are stored in a column vector. You may ask GAUSS to print the matrix using print:

Print vnames ;

This does not give you the right answer as GAUSS does not recognize you stored character instead of numerical data in vnames. To correct for this use:

Print $vnames ;

Rk2: the file does not look very good with capitalized and un-capitalized words but this does not matter as GAUSS "interprets" READR in the same way as readr or even REadR. GAUSS is **not case-sensitive**.

Rk3: The general syntax for OPEN is:

handle=fileName* FOR *mode* VARINDXI *offset ;

The *mode* is either 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 content cannot be read. UPDATE allows reading and writing.

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.

2.3 Closing and saving files

You can close a dataset using:
Close dataset ;
Close dataset1 dataset2 datasetn ;

You can save the content of a matrix into a GAUSS dataset using:
Export=saved(matrix,dataset,vnames) ;
Suppose the matrix is N*K. Then vnames has to be K*1. If dataset is null or 0, the data set name will be temp.dat. If vnames is a null or 0, the variable names will begin with “X” and be numbered 1 to K. The output data type is double precision. Export is a created variabole taking value 1 if the export is successful.

```
Let x={1 2,2.0000001 4.0000000000000000000001} ;
Let xnames={"X1","X2"} ;
Export=saved(x,"H:\\Antoine\\TEACH-EC475\\data\\example",xnames) ;
if export==0 ;
"the export is unsuccessful" ;
else ;
"the export is successful" ;
endif ;
```

3. Matrix manipulations

3.1. Creating and deleting matrices

a) Fixed values matrices

A=zeros(10,3) ; /* creates a matrix of 0s 10 rows * 3 columns */
I=eye(10) ; /* creates an identity matrix 10*10 */
B=ones(5,12) ; /* creates a matrix of 1s 5 rows * 12 columns */
C=seqa(*start,inc,n*) ; /* creates an additive column vector sequence of increment inc, starting at C1=start, C2=start+inc… and containing n values */
D=seqm(*start,inc,n*) ; /* creates a multiplicative vector sequence as seqa */

To save memory space inside a program you can clear some variables. That is set their contents to a 0 scalar. **clear B ;** It may seems strange that you are not able to delete the variables themselves. This can only be done at the start of a program using **new ;** or **delete all ;**. **Delete all ;** has the same impact as writing **new ;** at the start of your program. It clears every existing references. To keep tracks on things you may want to know what are the variables you created and their characteristics. You need to type:

Show all ; /* for all the variables */
Show/m all ; /* for all the matrices only */
Show A ; /* if interested in the var A */
Show var* ; /* for all the variables starting with var */

The user can also create a matrix using constants with the **LET** command:
LET x=1 2 3 4 ; /* column vector */

LET x=1,2,3,4 ; /* as before */
LET x=1 2,3,4 ; /* as before! */
LET x={1 2 3 4} ; /* row vector */
LET M={1 2, 3 4} ; /* 2*2 matrix with 1st row 1 2 */
LET M[2,3]=1 2 3 4 5 6 ; /* 2*3 matrix with 1st row 1 2 3*/
LET M[2,3]=1, 2, 3, 4, 5, 6 ; /* 2*3 matrix with 1st row 1 2 3*/
LET M[2,3]= 6 ; /* 2*3 matrix with all entries=6 */

b) "Random" matrices

N=RNDN(10,3) ; /* creates a 10*3 matrix of independent pseudo random draws of a N(0,1) */
S=1 ; N1=RNDNS(10,3,S) ; /* creates a 10*3 matrix of independent pseudo random draws of a N(0,1) with a given seed S=1*/
U=UNDN(10,3) ; /* creates a 10*3 matrix of independent pseudo random draws of a Uniform (0,1) */
S=12 ; U1=UNDNS(10,3,S) ; /* creates a 10*3 matrix of independent pseudo random draws of a Uniform (0,1) with a given seed S=12*/

Rk: the seed has to be an integer and allows you to repeat the same simulations.

Creating a vector of correlated random draws (B) from a normal random vector N(0,V) can be done using the Cholesky decomposition of the matrix V. With S the lower triangular matrix of the decomposition such that V = S*S', if U is a random vector N(0, Ik), then B=S*U has the required distribution N(0,S*Ik*S'=V). However as GAUSS returns the upper triangular Cholesky matrix, we have to adjust so that:

LET V={1 0.99,0.99 1} ;
Sl=chol(V)' ; /* Sl is Cholesky lower triangular V1/2 in some textbooks */
print V ;
print Sl ;
Seed=101 ;
U=rndns(2,100,Seed) ;
B = Sl*U ;
G=U|B ; /* vertical concatenation for comparison */
print /* U|B */ ; /* the second series of draws B1, B2 has values of B1 very close to B2 at each draw, while U1 and U2 are unrelated */
print G ;

Rk: Creating a GAUSS data-set from GAUSS generated matrices can be done in the same way as converting the matrix created from a text file into a GAUSS file (see part 2.1).

3.2. Operations on matrices[5]

a) Referencing matrices

This works in the same way for rows and columns:
g=mat[r1:r2,c1:c2] ; /* gives you rows r1 to r2 and col c1 to c2 of the matrix mat */
g=mat[r,c1:c2] ; /* gives you row r and columns c1 to c2 of the matrix mat */
g=mat[r,c1 c2 c3] ; /* gives you row r and columns c1, c2, c3 of the matrix mat */
g=mat[.,c1:c2] ; /* gives you all the rows and columns r1 to r2 of the matrix mat */

b) Simple operations

C=A*B ; /* product */
C=A' ; /* transpose */

Rk: Computing Z=X'X seems quite useful but GAUSS has an in built routines that is much faster:
Z=MOMENT(X,0) ; /* returns X'X ignoring missing values (1 or 2 instead of 0 delete them) */

C=inv(A) ; /* inverse */
C=invpd(A) ; /* inverse of a pos def matrix eg. X'X */
D=diag(A) ; /* diagonal of A */
D=det(A) ; /* det of A */
D=rank(A) ; /* rank of A */
C=cols(A) ; /* cols of A */
R=rows(A) ; /* rows of A */
E=eig(A) ; /* eigenvalues of A */

Pi is pi approx. 3.1415927.
^ is exponentiation
! is factorial
% is modulo division

C=A|B ; /* vertical concatenation of A and B */
C=A~B ; /* horizontal concatenation of A and B */

C=meanc(A) ; /* returns a column vector of the column mean of A */
Rk: Maxc(), Minc(),Sumc(), stdc() and **median()** do the same for the related operations.

C=A.*.B ; /* Kronecker product of A and B */
C=Vec(A) ; /* returns the column vector of the stacked columns below one another */
C=Vech(A) ; /* takes the lower triangular part of a symmetric matrix in a column vector */
Su=Chol(V) ; /* returns the upper triangular part of the cholesky decomp V=Su'Su */

[5] For a more extensive list by topics see:
http://www.timberlake.co.uk/software/gauss/400fns.pdf or the index of GAUSS help

It is also possible to sort a matrix with respect to a particular column (**sortc** for numeric values or **sortcc** for character values) or a group of columns (**sortmc**). The following illustrates the ranking of the values in GAUSS:

Let X={2 2 5 "E", . 1 6 "b", 2 1 3 "c", 3 2 7 "B", 4 3 8 "D"} ;
let mask=1 1 1 0 ; /* we define the columns of the matrix as num (1) or char (0) */
printfmt(X, mask') ; /* we print the matrix of char and num values */
X1=sortc(X,1) ; /* we sort wrt 1st column which contains numeric values */
X2=sortcc(X,4) ; /* we sort wrt 4th column which contains char. values */
printfmt(X1, mask') ; /* missing values are considered as arbitrarily small */
printfmt(X2, mask') ; /* GAUSS considers All capitalized letters as smaller as the un capitalized ones */
let sorter=1 2 ; /* we plan to sort the matrix wrt columns 1 2 of num. values */
X3=sortmc(X,sorter) ; printfmt(X3, mask') ;
let sorter=1 -4 ; /* we plan to sort the matrix wrt columns 1 4 of num. values and char. Values so we had a – before the 4 */
X4=sortmc(X,sorter) ; printfmt(X4, mask') ;

c) Operations element by element

C=sqrt(A) ; /* returns sqrt(aij) other functions are ln, log, exp, cos, sin */
C=A^2 ; /* returns aij^2 */
C=A./B ; /* returns aij/bij it works also with B a vector and gives a matrix aij/bi */
C=A.*B ; /* returns aij*bij it works also with B a vector and gives a matrix aij*bi */
Rk In general you can use the usual symbol but with a dot to do element by element operation. Eg. A.^B gives aij^bij.

C=A-3 ; /* returns aij-3 */
C=A+3 ; /* returns aij+3 */
C=A*3 ; /* returns aij*3 */
C=A/3 ; /* returns aij/3 */

d) Relational and logical operators

== or EQ for equality (this is different from A=5 where A is assigned value 5)
/= or NEQ for not equal
> or GT for greater than
>= or GE for greater or equal than
< or LT for lower than
<= or LE for lower or equal than

GAUSS returns 1 if a statement is TRUE and 0 if FALSE. You can combine statement using logical operators:

NOT Statement 1
Statement1 AND statement2
Statement1 OR statement2
Statement1 XOR statement2 (one, or the other, but not both)

Eg.
LET V={1 0.8,0.8 1} ;

LET V1={1 0.7,0.8 1} ;
C=V.==V1 ; /* this check element by element the == of the 2 matrices */
Print C ;
This returns:

```
    1.0000000       0.0000000
    1.0000000       1.0000000
```

In numerical computations, it is useful to be more tolerant on the meaning of ==. There is always some elements of rounding. You can set yourself the comparison with some tolerance level or use the **fuzzy comparison operators** of GAUSS. This corresponds to the usual operators with a prefix F (eg EQ is now FEQ). This can also be use for element by element comparison using the prefix DOTF (eg. DOTFEQ).

Eg.
A=1*10^(-15) ; B=1*10^(-16) ; TOL=1*10^(-15) ;
C=A==B ;
CT=A GE B-TOL AND A LE B+TOL ; /* this uses the logical operator AND !! */
_fcmptol=10^(-15) ; /* set the tolerance limit of gauss to 10^-15 this is the default value for the fuzzy comparison operators */
CTG=FEQ(A,B) ;
Print C ;; Print CT ;; Print CTG ;
Returns :

```
0.00000000       1.0000000       1.0000000
```

4. Procedures

4.1. Writing and using a procedure (PROC ENDP)

A procedure is a sub-program which will help you doing repeated tasks inside the same program. It has a number of inputs or arguments and can return several outputs. A simple procedure may have only one argument. For example, we can create a procedure that multiplies X' by X and return the determinant of this matrix.

proc(1)=detpd(X) ; /* (1) is the number of returned outputs */
local Y ; /* note that you have to write down the local var you are using in your proc this can be done in several steps as here or using: local Y, X ; */
local Z ;
Y=X'*X ; /* may be faster to use =MOMENT(X,0) */
Z=det(Y) ;
retp(Z) ; /* specifies the output of the proc */
endp ; /* note the syntax proc(#arg)=func() ; endp ;*/

X=ones(5,2) ; /* let's test the process */
d0=detpd(X) ;
print "the det of X'X is" ;; print d0 ;

Returns: the det of X'X is   0.00000000
If I now define:

X=eye(2) ; d0=detpd(X) ; print "the det of X'X is" ;; print d0 ;

This returns: the det of X'X is 1.0000000
A procedure can be much more complicated having several arguments and several outputs. The general syntax is:

PROC (# ***outParams*****) =** ***ProcName (inParam1, inParam2,... inParamN*****) ;**
LOCAL ***locVar1*** **; …**
LOCAL ***locVarN*** **;**
…
RETP (***outParam1, outParam2, ... outParamN*****) ;**
ENDP ;

You can then have access to the results using (for example):
{o1,o2,…oN}= ***ProcName (inParam1, inParam2,... inParamN*****) ;**
This allows to save the results of the procedure into matrices o1 to on that you can use later on. You could also use:
Call ***ProcName (inParam1, inParam2,... inParamN*****) ;**
Call allows procedures to be called when you do not need the return values (o1 to on). This is especially useful if a function produces printed output (ols for example).

4.2. Some existing procedures (OLS, DSTAT)

GAUSS has a built-in OLS procedure:
{vnam,m,b,stb,var,stddev,sig,cx,rsq,res,dwstat}=ols(0,y,x) ; /* see help OLS */
And a command giving some basic descriptive statistics.
{vnam,mean,var,std,min,max,valid,mis} = dstat(dataset,vars);

5. Looping (DO UNTIL, DO WHILE)

There are two main equivalent statements:

```
DO WHILE condition;          DO UNTIL condition;
  doSomething;                 doSomething;
ENDO;                        ENDO;
```

Eg.
X=0 ; I=0 ;
DO WHILE I LE 5 ; /* LE is used for <= // could use DO UNTIL I==5*/
X=X+I ;
I=I+1 ;
ENDO ;
Print “X=” ;; print X ;
Returns the sum of the integers from 1 to 5, 15.

6. Branching (IF ELSE ELSEIF…)

The syntax of the IF statement is:

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

An example in PS1 is to check that the explanatory variables include a constant:

```
LET X={1 2 3, 1 4 5, 1 6 2} ;
z=0 ;
i=1 ;
do while i<=cols(x) ;
if minc(x[.,i])==maxc(x[.,i]) ; /* check col I min= col I max */
    if minc(x[.,i])==1 ; /* check col I min= 1 */
    z=z+1 ;
    else ;
    z=z+0 ;
    endif ; /* rk we could have use AND rather than imbricate 2 ifs conditions*/
else ;
z=z+0 ;
endif ;
i=i+1 ;
endo ;
if z>0 ; print “THERE IS A CONSTANT” ;
else ; print “NO CONSTANT” ;
endif ;
```

7. Printing, saving and storing your results

7.1. Printing (PRINT, FORMAT)

As we have already seen we can print matrices and strings inside GAUSS output. Print in the basis of all the printing we have done. To summarize:

```
print “THERE IS A CONSTANT” ;
let varnames= {“X1” “X2” “X3”} ;
print $varnames ; /* returns X1 X2 X3 */
let x=”WORD” ;
print $x ;; /* ;; means print the next output at same level */
let x2=2 ;
print x2 ; /* need separate print statements as as soon as $, the text option, is used, it
is used for the remaining part of the print statement */
```

print x2 $x ;

format allows to define new options to print your results, such as the precision of the decimals. Eg.

x=6.888889999 ;
format /RD 6, 0; print x ;; /* use rd=right justified numbers 6,0= with 6 spaces for the number and 0 for decimals */
format /RD 6, 12; print x ;;
format /RDC 6, 5; /* rdC the delimiter between numbers is a comma */ **print x ;**
prints: 7 6.888889999000 6.88889,

7.2. More on printing (PRINTFM, PRINTFMT)

Printfmt(x,mask) allows to print matrices containing both columns of characters values and numerical values. Mask is a row vector of 0 (if the ith column is character) and 1 otherwise. If X is a N*K matrix then the MASK has to be a 1*K vector.
Let x={"A" 1, "B" 2} ;
Let mask={0 1} ;
Printfmt(x,mask) ;

```
Returns:    A        1
            B        2
```

Printfm(x,mask,fmt) allows you to combine this feature with more precise definitions of the formatting of each columns. Suppose that X is a NxK matrix.
Then the **Mask** is a 1*K vector containing ones and zeros which is used to specify whether the particular column is to be printed as a string (0) or numeric (1) value. With printfm the mask can also be a NxK matrix specifying if each element of the matrix is character or numeric.
The **Fmt** matrix is a Kx3 or 1x3 matrix where each row specifies the format for the respective column of x. The **fmt** contains 3 pieces of information: the format string, the width of the column (<80) and the precision (<17).
This can be best seen in an example.
Let x={"A" 1.5 0.7501221, "B" 2.35 0.60015, "C" 4.21 4.4899} ;
Then using **printfmt** will not help to correctly display the difference in precision in the last two rows.
Let mask={0 1 1} ;
Printfmt(x,mask) ;
Let fmt={"-*.*s" 8 8, /* first column format */
"*.*lf " 9 2, /* second column format */
"*.*lf " 9 7} ; /* third column format */
Printfm(x,mask,fmt) ;
Returns with printfmt:

```
    A        1.5      0.7501221
    B       2.35        0.60015
    C       4.21         4.4899
```

Or with printfm:

```
A        1.50  0.7501221
B        2.35  0.6001500
```

```
C          4.21  4.4899000
```

The following example defines all the elements in the matrix and do not proceed column by column. For example, this may be useful if the first row of the matrix contains the columns' names.

```
Let x={"VAR" "EST" "SE", "A" 1.5 0.7501221, "B" 2.35 0.60015, "C" 4.21 4.4899} ;
Let mask={0 0 0, 0 1 1, 0 1 1, 0 1 1} ;
Let fmt={"-*.*lg  " 8 8, /* first column format */
"*.*lf  " 16 3, /* second column format */
"*.*lf  " 16 7} ; /* third column format */
Printfm(x,mask,fmt) ;
```

Returns:

```
VAR          EST              SE
A            1.500      0.7501221
B            2.350      0.6001500
C            4.210      4.4899000
```

7.3. Saving the output of your results (OUTPUT)

You can decide to save the results of your output in a text file. Eg.

```
Output file= "H:\\Antoine\\TEACH-EC475\\prog_gauss\\gauss_outputs\\output1.txt" ON ; /* this opens the text file to store the results */
A={1 2 3} ; print A ;
Output off ; /* this closes the text file to store the results */
Output on ; /* this reopens the last the text file used to store the results and appends the current results to the past ones */
Output file= "H:\\Antoine\\TEACH-EC475\\prog_gauss\\gauss_outputs\\output1.txt" on ; /* Reopens the text file with the correct path if several output are used to store the results (on=append the new results to past ones) */
Output file= "H:\\Antoine\\TEACH-EC475\\prog_gauss\\gauss_outputs\\output1.txt" reset ; /* Reopens the last the desired text file used to store the results, clears it and reuses it to store the new results */
Output off ; /* this closes the text file to store the results */
```

A useful option is the length of the displayed lines in the output if we have matrices with a large number of columns and/or if we use a very precise format to display our results. This can be changed using:

OUTWITDTH *#char* ;

The #char has to be between 2 and 256 and the default value is 80.

7.4. Keeping track of time and dates

To keep a record of the date the following variables and functions are useful:

Date ; /* 4-element column vector, in the order: year, month, day, and hundredths of a second since midnight. */
datestrymd(date) ; /* function that returns time as yyyymmdd */
datestr(date) ; /* function that returns time as mm/dd/yy */
datestring(date) ; /* function that returns time as mm/dd/yyyy */
time; /* 4x1 numeric vector, the current time in the order: hours, minutes, seconds, and hundredths of a second.*/
timestr(time) ; /* function that returns time as hr:mn:sc */

Now if we want to know how long take our calculations we can check how much time different computations of Y'Y takes.

```
/* we define the variables */
let x=0 ;
clear y ; clear x ;
S=1 ;
y=ones(10^5,1)~RNDNS(10^5,20,S) ;

/* Method 1: using hsec */
et = hsec ; /* we record the start of the computation in hundred of seconds since midnight */
x = y'*y;
etp = hsec ;
etq= etp- et ;  /* we get an estimate of the duration of the computations */
print "time of *: (sec/100) ";; etq ;

/* Method 2: using the full date */
clear x ;
et1=date;
x = y'*y;
et2=date ;
et3 = ethsec(et1,et2) ; /* ethsec returns the # of hundreds of seconds between et1 and et2 */
"time of *: using ETHSEC (sec/100) " ;; et2 ;
```

This returns:

```
time of *: (sec/100)       9.4000000
time of *: using ETHSEC (sec/100)       9.4000000
```

8. References

1. Hill C. and Lee A., 2001, Using Gauss for Econometrics, 225p.
http://pages.suddenlink.net/ladkins/pdf/GAUSS.pdf
Contains a short introduction to Gauss and a lot of examples.
2. Gauss 9.0 documentation in pdf files, in particular the User guide (470p.).
http://seldon.it.northwestern.edu/sscc/gaussdoc/