Notes on Gauss

Joe Gochal[1]

GAUSS is a flexible and rather powerful statistical and mathematical program. By way of comparison, it is much more flexible and amenable to handling large and complicated data sets than Stata but is not as powerful and elegant as the S programming language (either in its S-Plus or R implementations). This middle of the road approach allows users to do things that aren't feasible (or in some cases possible) in more basic programs while avoiding the possibly large time investment required in learning S. However, GAUSS is not as straightforward to use as STATA or LIMDEP. These notes cover the basics of GAUSS. I do not include graphics because there are much better graphical packages out there than GAUSS, such as S-Plus and R.

To start, open GAUSS by clicking on the GAUSS 5.0 icon on the desktop (or open via Start>Programs>GAUSS50>Gauss 5.0 from the start menu. There are a number of ways to actually run and communicate with GAUSS. The easiest in Windows is through the command window. This window should be open when you open the program (if it isn't, go to "Windows" on the top toolbar and click on "Command Input-Output"). There should be a bunch of junk and something like:

To run an example program, enter the following at the command prompt:

run ols.e

Before we run this or any other program, we'll go over the basics how to input data and how to change the working directory (i.e., the place where everything goes via default). In general, to load in a space delimited ASCII file where each row is a separate observations and which is called, say, "d" with address p:\misc\data with R rows and C columns, we would type **load d[R,C] = p:\misc\d** where R and C are obviously numbers (note that for ASCII files there can be a .txt extension after the filename, so you should include it in the address if there is one)[2] and the stuff on the left hand side is the pathname of the data file. Using this command, GAUSS reads in the data row by row. Note that all this does is to make GAUSS read an ASCII file as a rectangular ASCII file. But we can also save this file as something called a matrix file (with extension .fmt). To do so, after having opened our ASCII file **d**, we just type **save p:\misc\d = d** which automatically be saved as **d.fmt** in the specified directory. The good thing about using matrix files is that once they are specified as such, we can load them without referring to their dimensions. Thus to load **d.fmt** in the future, we just type **load p:\misc\d** where there is no need to include the .fmt extension.

---

[1] Send questions, comments, and snide remarks, to jrg262@nyu.edu. Any and all errors are not the writer's fault.

[2] To read in a file with first row variable name headers, we could do the following: **load d[R,C] = p:\misc\d; vnames=d[1,.]'; d=d[2:C,.]; makevars(d,0,vnames);** See the GAUSS Language Reference for the **makevars** command**.** In this tutorial, we will use non-header data files.

The working directory is where output is saved if you do not specify a particular address. The default is in C:\GAUSS50. This can be changed. One way to do it on Windows machines is via an interface on the toolbar. Below File, Edit, … you should see a row of icons at the end of which that says C:\GAUSS50 followed on the right by a folder icon. If you don't see the toolbar, go to View and make sure "Main Toolbar" is checked. If you don't see the window and folder, go to View and make sure "Working Directory" is checked. To change the working directory, click on the working directory folder, and then choose what you want your working directory to be. If you know the address of the directory, it may be easier to change it via the **chdir h:**… ; command and hitting return (where h is just the drive you want it to go to and the … are where you put the address in that drive).

Typing in run ols.e will execute a program called ols.e from the c:\GAUSS50\examples file a print out some output. As a way of previewing how GAUSS works, it is worth looking at the program, which is printed below (it's case sensitive so beware):

```
rndseed 345346;
x = rndn(100,5);
y = rndu(100,1);

output file = ols_output.txt reset;

call ols("",y,x);

output off;

print;
print "Results of this run stored in ols_output.txt file";
print "The file will be found in your working directory";
```

The first thing to notice is that each line ends with a semi-colon (;), like the C language does, which serves the same function as the dollar sign ($) does in LIMDEP. To run any statistical routine, we need data, and in this example we create a random data set. **rndseed** just creates a seed for the following random number generation so don't worry about it. The next two lines create two matrices of (pseudo) random standard Normally distributed numbers. **x = rndn(100,5);** creates a 100 random observations of five normal variables, while **y = rndu(100,1);** does so for one random variable. Thus for the command **rndn**, the first element in the parenthesis tells GAUSS how many rows (observations) to create and the second element tells it how many columns (variables) to make. The next line, **output file = ols_output.txt reset;**, tells GAUSS where to put the output (in this case the present working directory) command). The main part of this program is **call ols("",y,x);** tells GAUSS to get its routine for estimating ordinary least squares, the **call ols** part, while the stuff in the parentheses tells GAUSS what dataset to use (the **""** says that it should use the stuff that was just created), what is the response variable (**y**), and what are the predictor variables (**x**). `output off;` tells GAUSS not to send the output of the regression to a another auxiliary file (using **on** instead of **off** and putting a **file** = directory path after output will create such a file). The print commands are self-explanatory.

Note that instead of entering each line in the command window, we could have hit CTRL-N to open an empty window in GAUSS and then written the entire program in that window, saved it, and then type **run** with the program's path and name. Thus we have something akin to log files in Stata and batch files in Limdep.

Let's keep the **x** and **y** variables created above and see how other things work. Say that we wanted to know the mean of the **x** variables. Do get this, we would type after a >> prompt **mx = meanc(x)**, where **meanc(x)** calculates the mean of the column vectors that comprise **x**, and **mx** is the name that is given to the vector or means created by the **meanc(x)** command (kind of like the objects created in S-Plus and R). Note that just entering **mx = meanc(x)** doesn't print out anything. To see the vector of means, we have to then enter **mx**. Until you tell GAUSS otherwise, **mx** will be a 5 x 1 vector or means. You can save this in the working directory (as a .fmt file) with the command **save mx** (specifying a path will save it some other directory). Other useful descriptive commands are **stdc** (standard deviation) and **medianc** (median). Like the summary command in STATA, the **dstat** command tells GAUSS to give you a bunch of summary stats (the syntax is **dstat("", y);** where the "" is the dataset holder).

Data transformations are really easy. Say we want absolute values of **y** (call it **aby**) instead of the generated values of **y**. To get these, we just type **aby = abs(y);** to create a vector that contains the absolute value of the values in **y.** We can then get the sin or **aby** by **saby = sin(aby);** , the Fast Fourier Transform or **aby** by **ffaby = fft(aby);**, or the natural log of **y** by **lny = ln(y);**. There are a ton of built in transformations so if you want one just look in the appendix of the manual or try the help menu (e.g., look up **corrm** in the help section to see how to do a correlation matrix).

Let's go back OLS. The **call ols("",y,x); command** gave us the basic output from an OLS regression but we often want more information (e.g., residuals, variance-covariance matrix, Durbin-Watson, R-squared, etc). To see how this is done, and how to estimate different models, we're going to use real world data, in this case from a data set on people aboard the Titanic. This should be stored on the lab's P drive as p:\misc\titanic.dat (variable descriptions are under titantic_disc.asc) which is a space delimited ASCII file. GAUSS can read in data in a number of ways but we are going to concentrate on reading in ASCII files for the time being.

For future reference, GAUSS has a host of matrix commands including: **rows(d), cols(d), det(d), diag(d), rank(d), inv(d), invpd(d),** and **eig(d)** which give the number of rows, number of columns, determinant, diagonal elements, rank, inverse, inverse of a positive definite, and eigenvalue of matrix **d**, when such operations exist (remember that all operations on some object **d** are of the form **x=operation(d)**). +,-, and * are used for matrix addition, subtraction, and multiplication (when possible) just as for scalars. The forward-slash / is used for matrix division which really means **x=operation(d/f)** is equivalent to the following for some appropriate matrices **d** and **f**: **finv = inv(f)**; **x = d*finv**. Finally, say we want to extract the column **j** from matrix **d** for all the rows **R** and we will call that column **Cj.** To do so, type **Cj = d[., j].** In general, then, by typing **X=d[1 2 ...,1 2 ...]** we extract the first whatever rows of **d** and the first whatever columns

to form a whatever by whatever matrix **X.** When the argument before (or after) the comma is only composed of the period, we are telling GAUSS to take all the rows (or columns) of matrix **d**. The operation ~ concatenates matrices and ' transposes them.

So, using the above stuff let's write our own bivariate OLS program, in general terms. Say we have a matrix **d** whose first column is our response variable which we will call **y**. Say our predictor variable **x** is in some column **pj** of **d** and that we also want to estimated a constant term. We are interested in the coefficient vector **b**, the vector of residuals **e**, the estimated variance of the residuals **s2** the variance-covariance matrix and standard errors of the b's (**vb** and **seb**), the **t** value of the the b's, **R2** calculates the r-squared statistics. In what follows, **new** is a control command that clears the previous junk that GAUSS had on its plate, **clear** clears previously loaded matrices, and **ones(R, C)** creates a matrix of ones of specified dimensions.

```
new;
load d[R,C] = h:\….;
y = d[.,1];
pred = d[., pj];
x = ones(R,1)~pred;
clear d;
n = rows(y);
k = cols(x);
b = invpd(x'x)*x'y;
e = y – x*b;
s2 = (e'e)/(n-k);
vb = s2*invpd(x'x);
seb = sqrt(diag(vb));
tb = b./seb;
R2 = 1-e'e/(y-meanc(y))'(y-meanc(y));
end;
```

You can run this by saving it as its own program as mentioned earlier or by copying and pasting it into GAUSS after a >> prompt and then hitting return. After running the program, making sure to put a full output file address, typing b, for example, will give you the estimated regression coefficients, with the first being the constant and the second being for the predictor variable (as is specified in our definition of **x**). Obviously, to make this a multivariate regression, we just need to have **pred = d[.,pj1 pj2...].**

This example should serve to highlight the relative east with which GAUSS can handle matrices. Let's use the titanic data set. We are going to use the fare of the ticket (the 6^{th} column) as our response variable and age (the 3^{rd} column) as the predictor variable. First, note that the above OLS program is easily adaptable for the present case. It would look like the following (put in appropriate addresses):

```
new;
load d[1308,6] = p:\….titanic.txt;
y = d[.,6];
pred = d[., 4];
x = ones(1308,1)~pred;
clear titanic;
n = rows(y);
k = cols(x);
b = invpd(x'x)*x'y;
e = y – x*b;
s2 = (e'e)/(n-k);
vb = s2.*invpd(x'x);
seb = sqrt(diag(vb));
tb = b./seb;
R2 = 1-e'e/(y-meanc(y))'(y-meanc(y));
end;
```

To run this as a via the **call ols** command, we would type the following code. Note that we do note need to specify a placeholder of the constant since **call ols** will automatically estimate a constant. The command __altname renames the variables **X1, X2**, … to the names in quotes for the purposes of the OLS output.

```
new;
load d[1045,6] = p:\misc\titanic;
yc = d[.,7];
xc = d[., 4];
output file = ctitols.txt, reset;
__altname = "Constant"|"Age";
call ols(0,yc,xc);
end;
```

After running this program, the screen (output file) should have a regression output table that looks very much something you would see in STATA, with a bunch of regression summary statistics on top followed by estimates of coefficients, standard errors, t-values, and whatnot. See the GAUSS Language reference manual to see more about this built in command.

As stated earlier, if all you wanted to do was to run simple stuff like OLS, there would be no reason to use GAUSS. However, if you wanted to do more interesting stuff, then GAUSS is better suited to original programming than, say, STATA. Many GAUSS programs are implemented in the procedure, or "**proc**," environment. Basically, for GAUSS a procedure is a group of statements in a self-contained form that takes in inputs and gives out outputs – i.e., it is a user defined program. Since GAUSS has relatively little in the way of canned procedures, the **proc** environment is very useful. We'll spend some time looking at it. Note that GAUSS also can be used to run things called functions which are like procedures but only compute a single expression and give only one output variable. Functions are denoted by **fn**.

Some people love writing GAUSS procedures. For the most part, they are very simple to do. Take the following procedure written by Gary King (http://gking.harvard.edu/gauss/cdfnorm.g) that calculates unstandardized normal CDF's.

```
**  (C) Copyright 1999 Gary King
**  All Rights Reserved.
**  http://GKing.Harvard.Edu, King@Harvard.Edu
**  Department of Government, Harvard University
**
** unstandardized cumulative normal distribution
**
** p = cdfnorm(y,mu,sigma2);
**
** INPUTS:
** mu = mean
** sigma2 = variance
** y = normal variate
**
** OUTPUT:
** p = Prob(Y<y|mu,sigma2), where y is a realization of the r.v. Y
**
*/
proc cdfnorm(y,mu,sigma2);
  local sigma;
  if sigma2<=0;
    "cdfnorm: variance must be positive";
    end;
  endif;
  sigma=sqrt(sigma2);
  retp( cdfn((y-mu)./sigma) );
endp;
```

After writing an running this procedure, we would call it by typing **p = cdfnorm(y,mu,sigma2);** using numeric values for **y**, **mu**, and **sigma2** to get the specified probability. Thus **cdfnorm** is the name of our new little program, with the required inputs as listed. The **local** line just specifies "local" variables that are to be used internally by the procedure, i.e., those not defined before the procedure, included as input, or used after the procedure. Note the **if** command which terminates the procedure in an error is the inputted variance is negative and goes on to the next commands if the variance is kosher. It then calculates the standard deviation from the given variance. The **retp** line tells GAUSS what the output of the procedure should be – in this case, the

Prob(Y<y|mu,sigma2). If there we more than one output for the procedure, we would include multiple arguments in the parentheses. The command **endp;** formally ends the procedure.

This trivial example highlights the basic structure of a procedure. In general, they look something like this:

```
proc NAME(INPUT);
local  VAR1, VAR2,….;
COMMANDS;
retp( OUTPUT)
endp;
```

Given this basic set-up, it's easy to turn the OLS programs written earlier into their own procedures. As a further example, the following procedure produces an *i* by *j* matrix of bivariate standard-normal random numbers. Note that **cov** is an *i* x *i* matrix of variances and covariances.

```
proc rmn(i, j, cov);
local x;
x = RNDN(i, j);
retp(x*CHOL(cov));
endp;
```

A good way to learn GAUSS is to start writing procedures for simple things and then move on ones more suited to your more problem-specific procedure. However, many interesting statistical questions in political science involve some sort of maximizing routine, most obviously maximum likelihood. There are number of ways to implement ML in GAUSS. We will focus on something called the Maxlik library, an add-on to the standard GAUSS program. GAUSS has many libraries, each of which is basically just a set of pre-written **proc** statements. Other powerful libraries include CML, or Constrained Maximum Likelihood, to perform ML in the face constraints on the parameters, CO, Constrained Optimization, which deals with constrained non-linear programming (which is very cool), Quantal Response which provides another way to estimate models like logits, probits, and whatnot.[3] To the best of my knowledge, the Politics lab computers only have Maxlik and CML, so if you have a problem that requires another library, ask the computer help people to see if they can get additional libraries.

To use Maxlik commands, we first have to call the Maxlik library via the command `library maxlik;` and then load the data as above and specify some output file. After that simple step, the commands for estimating ML procedures are rather straightforward. Gary King's Maxlik files[4] provide some nice examples. To further highlight the GAUSS programming, the rest of this section will use Maxlik to explore numerous aspects of a

[3] See http://www.scientific-solutions.ch/tech/gausslib/indexlib.html for a complete listing.
[4] http://gking.harvard.edu/stats.shtml#maxlik

logit model. Note that Maxlik is rather powerful so if can do more than just the simple stuff outline below. The best way to learn it is to play around with it using questions that are interesting to you.

Assuming that you call Maxlik and have loaded your data, the following procedure will calculate log likelihoods which will then be used to estimate a model , where the data is called "data," "b" are the coefficients, "x" the predictor variables, and "y" the binary response variable.

proc loglik(b, data)

```
local k, x, y, llhood, cdf;
k = cols(data);
x = data[.,1:k-1];
y = data[.,k];
cdf = 1./(1+exp(-x*b));
llhood = y.*ln(cdf)+(1-y).*ln(1-cdf);
retp(llhood);
endp;
```

In general, a Maxlik proc is a function of two arguments – the first being the parameters of the model (the "b" above) and the second being the data for which the log likelihood is to be calculated.

We could also write a procedure to derive the score matrix for a logit model:

proc score(b, data)

```
local k, x, y, s cdf, pdf;
k = cols(data);
x = data[.,1:k-1];
y = data[.,k];
cdf = 1./(1+exp(-x*b));
pdf = cdf./(1-cdf);
s = y.*(pdf./cdf).*x - (1-y).*(pdf./(1-cdf)).*x;
retp(s);
endp;
```

Similarly, a Hessian for a logit could be produced via:

proc Hessian(b, data)

```
local k, x, y, pdf, hess;
k = cols(data);
x = data[.,1:k-1];
y = data[.,k];
pdf = exp(-x*b)./(1+exp(-x*b))^2;
s = y.*(pdf./cdf).*x - (1-y).*(pdf./(1-cdf)).*x;
retp(s);
endp;
```

Now, let's return to the first of these three procedures, **loglik**. This specifies the log-likelihood of the model. We use this is in something called the **proc maxlik** (i.e., maxlik procedure) to actually estimate our models. **proc maxlik** is a canned procedure in

Maxlik that performs the numerical optimization that we want. Formally, **proc maxlik** looks something like this:

```
{ b,f,g,cov,retcode } = MAXLIK(dataset,vars,&fct,start)
```

The left hand side variables in the curly brackets are the output of this procedure: "b" is the kx1 vector of estimated parameters (i.e., coefficients), "f" is the value of the log-likelihood, "g" is the kx1 vector of gradients evaluated at the "b" values, "cov" is the kxk covariance vector of the parameters, and "retcode" is scalar that equals zero if everything converges nicely and equals some other number if something goes wrong. For the sake of completeness, "retcode" has the following values for the following problems:

```
 1   forced exit
 2   maximum number of iterations exceeded
 3   function calculation failed
 4   gradient calculation failed
 5   Hessian calculation failed
 6   step length calculation failed
 7   function cannot be evaluated at initial parameter values
 8   number of elements in the gradient vector inconsistent
     with number of starting values
 9   gradient function returned a column vector rather than
     the required row vector
10   secant update failed
11   maximum time exceeded
12   weights could not be found
20   Hessian failed to invert
34   data set could not be opened
99   termination condition unknown
```

The stuff in the curly brackets following MAXLIK are the inputs to the procedure: "dataset" specifies the RxC data, "vars" is the vector of variables, "&fct" specifies what likelihood procedure (e.g., loglik above for a logit), and "start" is a vector of starting values for the maximization procedure (e.g., a vector of zeros or OLS coefficients). There are also a number of options that the user can fiddle with – for example, changing the optimization procedure, different convergence criteria, estimating heteroscedastic variance). If you're doing anything non-standard, you should acquaint yourself with them.

Thus the **proc maxlik** is a very powerful and general tool. Just give it some define log-likelihood function, and it will try and estimate it for you. Given these estimates, we could use them in the **score** or **Hessian** procedures define above to compute these matrices for a logit.

To practice with the Maxlik library, try implementing a probit, Weibull, Cauchy, Tobit, or Heckman model. Just write down the likelihood as I did for the logit and then run **proc maxlik** with the appropriate commands. Or use the Titanic data to estimate a model concerning survival (column 1). Obviously, we could do things like this in STATA. But given GAUSS's greater computing power, if you want to do a non-standard estimation, GAUSS's maxlik is the way to go.