

# A Brief Introduction to GAUSS

by David Aadland

---

## I. Getting Started

### A. Loading GAUSS

Gauss 5.0 for Windows is available in the graduate computer lab, Ross Hall 261. Simply go the “Start” button on any of the computers and proceed to “All Programs”. The Gauss icon should be clearly displayed, as well as a shortcut to the “online” gauss manual. Gauss Light, the free student version of Gauss, is also available. The limitations of Gauss Light are as follows: 10,000 elements and/or 100 X 100 matrix limit, 1 mb global symbol memory, 1 1/2 mb total workspace, and no source level debugger.

### B. User Interface

There are three basic windows in GAUSS: Command, Edit and PQG. The **PQG window** is the graphics display window that is automatically displayed during the course of a program when a graph is created. We will not directly discuss this window here—refer to the GAUSS *Supplement for OS/2 and Windows* for more details.

The **Command window** is where all the action takes place (i.e., input/output). A prompt (>>) appears when the window is first opened. You can type commands in directly at the prompt, hit enter, and they will be evaluated. However, this should only be done if you do not wish to save your results.

The last window, the **Edit window**, is where your programs will be composed. It is a blank window with an appearance much like the Command window except for the lack of a prompt. The primary advantage of composing programs within the Edit window is that if mistakes are made the commands may be edited and rerun without having to retype the commands. Alternatively, in the Command window, previously executed commands are no longer available and can only be accessed by retyping them in or by cutting and pasting.

## II. Working with Matrices

### A. Building a Matrix

Begin by loading GAUSS and opening the Edit window. The Edit window is opened by clicking on the Edit button at the bottom of the screen. Now let's build a 3×3 matrix by typing in the following:

```
A = {1 4 0, 4 3 8, 7 1 9};  
print A;
```

Now save the file as "temp" on the a: drive (or whatever drive you like). Once the file has been saved, click on the "Run Current Program" icon at the top of the screen or click on Action -> Run Program. Next let's build a  $3 \times 1$  column vector B:

```
B = {1,2,3};  
print B;
```

and a  $1 \times 3$  row vector C:

```
C = {4 5 6};  
print C;
```

## B. Matrix Functions

Although GAUSS has several hundred matrix functions, there are a few that you will probably find yourself using most frequently. Begin by finding the transpose of matrix A and, for comparison, printing both the transpose of A and A itself:

```
trA = A';  
print trA; A;
```

Next, let's find the inverse of A:

```
invA = inv(A);  
print invA;
```

To check whether the inverse has been calculated correctly, let's matrix multiply A by invA, which should give us the identity matrix:

```
id = A*invA;  
print id;
```

Notice that the off-diagonal elements are slightly different than zero. This is due to rounding error and will not substantially alter any calculations. Now let's redefine the identity matrix, create a new matrix of ones, create a new matrix of zeros and print them out with labels:

```
id = eye(3);  
unity = ones(3,3);  
null = zeros(3,3);  
print "id = " id; "unity = " unity; "null = " null;
```

Often it will be necessary to generate a matrix of random numbers. First, let's create a column vector of *uniformly* distributed random numbers; and second, let's create a row vector of *normally* distributed random numbers:

```

D = rndu(3,1);
E = rndn(1,3);
print "vector of uniformly distributed random numbers = " D;
    "vector of normally distributed random numbers = " E;

```

Sometimes you will need to generate an additive or multiplicative sequence of numbers. First, let's create a column vector beginning with 0 and ending with 20, increasing by increments of 2. And second, let's create a 10×1 column vector beginning with 2 and increasing by multiples of 2:

```

F = seqa(0,2,11);
G = seqm(2,2,10);
print "F = " F; "G = " G;

```

### C. Extracting Submatrices

Often it will be necessary to extract a row(s) or a column(s) from a matrix. Let's extract both the first row and the second column from matrix A:

```

r1A = A[1,.];
c2A = A[.,2];
print "A = " A; "first row of A = " r1A; "second column of A = " c2A;

```

Now let's extract the 2×2 submatrix in the lower right corner of A:

```

lowrightA = A[2:3, 2:3];
print lowrightA; A;

```

And finally, let's extract the 2×2 matrix comprised of the intersection of rows 1 and 3 and columns 1 and 2:

```

subA = A[1 3, 1:2];
print subA; A;

```

### D. Matrix Operations

Begin with element-by-element operators. Let's first add the identity matrix to A and then subtract the unity matrix:

```

H = id + A;
print A; H;
J = H - unity;
print J;

```

Next, let's multiply each element of H by each corresponding element of J:

```
K = H .* J;
print K;
```

Element-by-element multiplication allows you to multiply together two matrices that would not be conformable under normal matrix multiplication. To see this, multiply the (3 x 3) matrix A by the (1 x 3) row vector r1A using both matrix and element-by-element multiplication:

```
L = A.*r1A;
M = A*r1A;
print A;
print;
print r1A; L;
```

As expected, GAUSS returns a "Matrices Not Conformable" error. Now delete the matrix multiplication command and rerun. Notice that the (1 x 3) vector is swept down the (3 x 3) matrix A. The extension to column vectors and element-by-element division is straightforward.

Next consider normal matrix multiplication and division, rather than element-by-element operation. Calculate the square of A and then retrieve A by dividing the square by A:

```
AA = A*A;
print AA;
A1 = AA / A;
print A1; A;
```

Several other matrix operators that may be of interest are: horizontal and vertical concatenation, and Kronecker products:

```
AhorA = A~A;
print AhorA;
AverA = A|A;
print AverA;
AkronA = A .* A;
print AkronA;
```

### **III. Working with Data**

#### **A. Loading Data**

ASCII data can be loaded into GAUSS using the command **Load**. Let's begin by loading the sample data set "table2.1" found in our class webpage:

```
load path = c:\temp\;
load x[10,5] = table2.1;
```

Now take a look at the data and delete the first row, which contains the labels for the variables.

```
print x;
data = x[2:10,.];
print data;
```

The data are now loaded into a matrix in which columns 2 through 5 contain the observations on a single variable. Column 1 contains the dates, which range from 1972 through 1980.

## **B. Saving Data**

The command **Print** will allow us to write a matrix to disk. Let's save columns 2 and 3 of the matrix we loaded into GAUSS and save it to a new ASCII file.

```
format /rd 8,4;
output file = candy.dat reset;
screen off;
candy = data[:,2:3];
print candy;
screen on;
```

You can also save matrices as GAUSS data sets, which allow extremely fast reading and writing of data. Many of the library functions require the data to be stored in GAUSS format. GAUSS data sets are organized in terms of rows and columns, where each column is assigned a name. Let's save the same two columns above as a GAUSS data set:

```
dataset = "a:\gcandy";
vnames = {"C" "Y"};
y = saved(candy,dataset,vnames);
```

## **IV. Graphics**

GAUSS has several routines for creating and editing graphs. You can build Cartesian graphs, 3-D surface graphs, bar graphs, histograms and contour plots to name a few. Let's build two different types of graphs: (1) an xy "Cartesian" graph and (2) a histogram.

First, you need to activate the pgraph library and reset all graphics global variables (not always necessary, but good practice).

```
library pgraph;
graphset;
```

Start with the xy graph. Let's graph consumption (2<sup>nd</sup> column of x) versus time.

```
title("Consumption from 1972 through 1980");
xy(data[:,1],data[:,2]);
```

Next, lets build a histogram from a vector of randomly generated numbers.

```
xu = rndu(100,1);
title("Sample Uniform[0,1] pdf");
{b, m, f} = hist(xu, 20);
```

## **V. Procedures**

Procedures are user-defined functions that allow frequently used routines to be easily called up and evaluated without having to rewrite the code for the routine each time you wish to use them. While procedures can read global variables, they rely on local variables that can be read inside but not outside the procedure. Let's write a simple procedure that subtracts the mean of any given series. First, save the current file, and then, open a new Edit window and type in the following procedure:

```
proc (1) = demean(x)
    local(nobs, xbar)
    xbar = meanc(x);
    retp(x - xbar);
endp;
```

Save the file as "demean.src" in the gauss directory. Next, open "a:\temp" again and make this procedure available by typing the following:

```
lib user demean.src
```

which adds the procedure to the default library user.lcg. Now delete the previous command and call the procedure:

```
consdm = demean(data[.,2]);
print consdm; data[.,2];
```

## **VI. Miscellaneous**

### **A. Strings**

Strings can easily be created in GAUSS and attached to one another to form string arrays using string concatenation. They are especially useful for improving the appearance of output produced by the print statement.

```
s1 = "The A matrix ";
s2 = "is equal to ";
s = s1 $+ s2;
print s A;
```

### **B. Looping**

The primary command for looping is **Do**. Coupled with **While** or **Until**, the Do loop will execute a body of statements While the statement is true or Until the statement is false. Two other commands, **Continue** and **Break**, control the execution of the loop. When Continue is encountered, the Do loop will immediately jump back to the While or Until statements. When the Break statement is encountered, the Do loop is immediately terminated and the subsequent command is evaluated. Here is an example of a Do loop that calculates and records the sample means of ten  $N(0,1)$  random samples, each of size 100:

```
loops = 10;
means = zeros(loops,1);
counter = 1;
do while counter <= loops;
    x = rndn(100,1);
    avg = sumc(x)/100;
    means[counter,1] = avg;
    counter = counter + 1;
endo;
print means;
```

Another looping method involves the **For** statement. Below is an alternative method of calculating and recording a vector of sample means:

```
loops = 10;
means = zeros(loops,1);
for ii(1,loops,1);
    x = rndn(100,1);
    avg = sumc(x)/100;
    means[ii,1] = avg;
endfor;
print means;
```

### C. Relational and Logical Operators

There are several relational operators: less than ( $<$ ), greater than ( $>$ ), less than or equal to ( $<=$ ), greater than or equal to ( $>=$ ), equal to ( $==$ ), and not equal to ( $\neq$ ). The result will be either a zero or one depending upon whether the comparison is false or true respectively. By preceding the above relational operators with a dot ( $.$ ), the result will be a matrix of zeros or ones based on an element-by-element comparison.

```
A==A;
A<A;
A<=A;
A/=(A+A);
A.==(A+id);
```

## **D. Syntax**

There are several useful rules of syntax to remember: (1) GAUSS is not case sensitive except inside double quotes; (2) comments can be inserted into the program as follows: `/* Comment */`; (3) all statements in GAUSS should end with a semicolon (`;`); and (4) column position is unimportant and blank lines are allowed.

## **E. Operator Preference**

The order of operator preference is (from highest to lowest precedence):

1. exponentiation (^)
2. factorial (!)
3. multiplication (\*) and division (/)
4. addition (+) and subtraction (-)