NOTES ON GAUSS

Marc Nerlove
Department of Agricultural and Resource Economics
University of Maryland
Tel: (301) 405-1388 Fax: (301) 314-9032
email: mnerlove@arec.umd.edu
homepage: http://www.arec.umd.edu/mnerlove/mnerlove.htm

Omnia disce, videbis postea nihil esse superfluum.

"Why," said the Dodo, "the best way to explain it is to do it. (And, as you might like to try the thing yourself, some winter day, I will tell you how the Dodo managed it.)
Lewis Carroll, *Alice's Adventures in Wonderland*, 1865

The following notes are adapted from the GAUSS manual, *GAUSS, Mathematical and Statistical System: Supplement for OS/2 and Windows; Vol.1, System and Graphics Manual; Vol. 2, Command Reference,* Maple Valley, WA: Aptech Systems, 1984-96. I make use of a series of small GAUSS programs, basic01.gss through basic26,gss, to illustrate commands and manipulations. The programs and the output from them are reproduced as an appendix to these notes. The programs and the data necessary to run them are available at the homepage for this course: **http://www.arec.umd.edu/gauss/gauss.htm.**

If you are working on the AREC network, you will need to establish a working directory on your local machine. GAUSS is k:/Gauss32/gauss.exe. You can set up a shortcut with k:/Gauss32/gauss.exe as the target; the working directory is whatever you fill in the "Start in" box. I use c:/files/gaussfls for example. **GAUSS does not recognize file names of more than 8 characters.** You can also set up folders and direct output other than graphics to specific locations on your local machine. Graphics are not specifically addressable; graphics output will appear in your local directory with a tfk tag.

CONTENTS

5. MATHEMATICAL AND STRING OPERATORS

6. RELATIONAL AND LOGICAL OPERATORS FOR SCALARS OR MATRICES

7. PROGRAM FLOW CONTROL: DO-LOOPS, FOR-LOOPS, AND BRANCHING
8. PROCEDURES, FUNCTIONS, KEYWORDS AND LIBRARIES

9. SIMPLE GRAPHS IN GAUSS

10. MATHEMATICAL FUNCTIONS IN GAUSS: PI, TRIGONOMETRIC AND OTHER FUNCTIONS
    a. Trigonometric Functions
    b. Other Mathematical Functions

11. CHARACTERISTIC ROOTS AND VECTORS; MATRIX DECOMPOSITIONS; SOLUTION OF LINEAR EQUATIONS.

12. POLYNOMIALS.

13. FOURIER TRANSFORMS

14. RANDOM NUMBER GENERATION AND STATISTICAL DISTRIBUTIONS
    a. Random Number Generators
    b. Density Functions
    c. Cumulative Distribution Functions

15. NUMERICAL DIFFERENTIATION AND INTEGRATION
    a. Numerical Differentiation
    b. Numerical Integration

16. MAXIMIZATION AND MINIMIZATION: **QNewton** and **QProg**
    a. Examples Comparing Steepest Ascent with QNewton
        (1) Poisson Count Regression
        (2) Regression with First-Order Serially Correlated Disturbances
        (3) Regression with Heteroskedastic Disturbances
        (4) Regression with Box-Cox Transformations
        (5) Nonlinear Regression
    b. An Example of Quadratic Programming using Qprog

1.GAUSS/WINDOWS INTERFACE; BASIC STRUCTURE OF A GAUSS PROGRAM

When you click on the GAUSS shortcut you will bring up the COMMAND window.

COMMAND WINDOW
The Command window is the window that first appears when you start GAUSS. Its main feature is a large scrollable text edit pane. This is where the GAUSS prompt appears for you to issue interactive GAUSS commands**. It is also where all GAUSS program I/O takes place.**

Your GAUSS **help** file is located in k:/Gauss32/gauss.hlp. When you click on help the first time it, GAUSS may ask you to locate this file for it because your working directory is established in another location than the GAUSS exe. You can make a copy of **gauss.hlp** to put in your GAUSS working directory. Other menu items are: File, Edit, Search, Font, Options, Window, and Help.

File | Edit...
Opens a dialog for selecting a file to edit. The dialog lets you navigate drives and directories, specify a filename pattern, and see the files in a directory. The file selected is added to the top of the Edit list and loaded into the Edit window, which is brought to the foreground.
File | Run...
Opens a dialog for selecting a file to run. The file selected is added to the top of the Run list and the file is executed, bringing the Command window to the foreground.
The **debug** option is not yet implemented in the Windows version of GAUSS.
File | Stop...
Stops the currently executing GAUSS program. This can be very valuable if you've written a never-ending do-loop.
File | Exit GAUSS...
Exits GAUSS without verification. If you merely close the window GAUSS will ask you if you really want to go.
Edit | Undo Cut Copy Paste
Standard windows commands.
Search | Goto Find Replace Find Again Replace Again
Standard windows commands.
Font | Select...
Opens a font selection dialog which allows you to change the font in the text edit pane of the Command and Edit windows. The font setting is saved between GAUSS sessions. The default is not very convenient. I recommend a small compact font.
Options | Edit...
Includes controls to: Set the tab length. Set insert/overstrike mode.Turn text wrapping on/off.
Options | Program...
Includes controls to: Turn default libraries on/off. Set autoload/autodelete state. Turn declare warnings on/off. Turn dataloop translation on/off. Turn translator line tracking on/off. Turn normal linetracking on/off. Set Compile to File state. I will deal with these later. For now, leave the default settings.
The **debug** option is not yet implemented in the Windows version of GAUSS.
Windows simly lets you move between the Command and Edit windows.

The COMMAND window will accumulate output unless you make some provision to clear the screen. **There is no hot key for doing so.** You can to so when a GAUSS programs runs successfully by using a **cls** command near the beginning of the program. To do so within the COMMAND window requires typing **cls** after the GAUSS prompt and hitting enter.

EDIT WINDOW
The Edit window is a text editor. The Edit window and the Command window have many controls in common. The most obvious difference is that the Command window has Stop (running program) controls and the Edit window has Save (file) controls.

File | Save as...
Opens a dialog for saving the current contents of the Edit window under a new filename. The text being edited is saved to the new file, and the new filename is added to the top of the Edit list. You will find this useful for copying a program under a new name so you can use it as a template for creating a new program.

When you have run a program which creates a graph, GAUSS will open a PQG window. I will deal with this window and its options below. When GAUSS creates a graph it stores it in a file with a .tkf extension. **Unless you specify a new name for each graph you create, GAUSS will overlay the window and destroy the previous graph. tkf files are non-addressable; they will be stored in your general GAUSS working directory. If you have not established such a directory off the k-drive, to which you are not allowed to write, NO GRAPHS CAN BE CREATED.** After you have created a graph you can save it and protect it from overlay by moving it to a folder or separate directory.

I will deal with basic data input and output below, as well as with formating statements. You should create and save all your programming work in the edit window and use the command window only to view your output, or, occasionally, to check the value of a parameter you have not outputted to the command window. I strongly recommend that all your programs follow a common format which is outlined in the sample program, **basic01.gss**, reproduced in the appendix to thse notes.

2. CREATION OF VECTORS AND MATRICES. DATA INPUT AND OUTPUT.

There are basically two ways to get data into GAUSS: (1) You can enter vectors and matrices directly into your program, or (2) you can read the data in from a file. Both methods have their uses. You get data and results out via your COMMAND window or your output file by using the **print** command.

a. Direct Creation. Uses of **let.**

This command initializes matrices, string arrays, and strings. You can use a simpler format without **let**; with let the following options are available:

```
let [type] x = { const_list };
let [type] x[r,c] = const_list;
let [type] x[r,c] = { const_list };
let [type] x = const_list;
```

where

type = optional literal, symbol type, can be matrix or string**. If not supplied, matrix is assumed.**
const_list = list of constant values, numeric or character data for matrices, string literals for string arrays and strings.
r = scalar, the number of rows x will have.
c = scalar, the number of columns x will have.

The most common usage is

let x = { const_list };

x will have as many elements as are listed in const_list. **Elements within a row are separated by spaces, rows are separated by commas.** List the elements in row major order. Some examples are:

2x2 Matrix	let matrix x = { 1 2, 3 4 };
2x2 Character Matrix	let vars = { age pay, sex occ };
2x2 String Array	let string vars = { age pay, sex occ };
Column Vector	let x = { 1, 2, 3, 4 };
Row Character Vector	let x = { age pay sex occ };

Note that the syntax for creating string arrays is identical to the syntax for creating matrices; the string keyword is what determines whether the result will be a matrix or a string array.

The let is optional when you use curly braces ({ }). For example:

```
x = { 1 2 3, 4 5 6 };
```

See GAUSS help for a description of the other options.

Complex numbers can be entered by joining the real and imaginary parts with asign (+ or -); **there can be no spaces between the numbers and the sign.** Numbers with no real part can be entered by appending an "i" to the number. E.g.,

```
let x = 1.2+23i  8.56i  3-2.1i  -4.2e+6  5.46e-2i;
```

Numbers can also be entered in scientific notation, as indicated. This notation is described below.

Another way to dimension a matrix before filling it up, for example in the program itself, is to use the **zeros( )** or the **ones( )** matrices.

b. Reading Data from Previously Created Files. Uses of **load.**

The command **load** can be used to read in data from a an ASCII file (*.asc, *.txt, *.prn, or *.csv tag) or a GAUSS data file (.fmt tag). This command loads a matrix from an ASCII matrix file or a .fmt file. ASCII files must be delimited with spaces, commas, tabs or newlines. If your data is in an EXCEL file or can be put into an EXCEL file, you can save it as a tab delimited file (*.txt) or a space delimited file (*.prn) or a comma delimited file (*.csv). Then use the form

load y[ ] = filename.extension;

This loads the ASCII file named filename into the matrix y. If no dimensions are specified, y is a column vector containing all of the data in filename in the order 1st row, 2nd row, etc. (row major order). You can use the function **rows(y)** to find out if all the data have been loaded and the function **reshape (y, n, k)** to reshape the nk by 1 vector y into the matrix you want. More on **reshape** below. Alternatively, if you have an ASCII data set which is already in matrix form, n by k, you can simply use the command

load y[n,k] = filename.extension;

directly. I find the comma delimited form the most convenient. (Note that, in a **load** statement, the indices n and k refer to the total number of rows and columns to be in y, not the coordinate of an element.) On the other hand, the command

load x;

loads the file x.fmt into the matrix x. *.fmt files are created in GAUSS using the **save** command. Although GAUSS data file are more efficient that ASCII files, to save them you have to specify a path. It is easier to keep files you use in comma, tab or space delimited form. You can output matrices from GAUSS using the **print** command in *.txt form and go from there, e.g. via EXCEL.

c. Outputting Data with **print. format** Statements.

GAUSS provides for a number of different formats to be used together with the **print** command to output data to the command screen and/or the output file you specify. Both are text files and you can cut and paste from there. Note: **print does not print to a printer but rather to your COMMAND screen and stores output in your output file.**

The **format** comand sets the output format for matrices, string arrays, and strings printed out with **print**. The **format** command is formatted as follows:

format [/typ] [/fmted] [/mf] [/jnt] [fld,prec];

where

/typ	symbol type flag.
/fmted	enable formatting flag.
/mf	matrix row format flag.
/jnt	matrix element format flag -- controls

justification, notation and trailing character.

fld scalar, element field width.

prec scalar, element precision.

/typ options:

/mat Indicate which symbol types you are setting the output format for.

/sa Formatting parameters are maintained separately for matrices (/mat),

/str string arrays **(/sa**), and strings **(/str**).

You can specify more than one **/typ** flag; the format will be set for all types indicated. **If no /typ flag is listed, format assumes /mat.**

/fmted options: **<u>You should ignore this option</u>**

/on, **/off** Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped to the screen in a "raw" format.

/mf options:

/m0 print no delimiters before or after rows of a matrix

/m1 or /mb1, print 1 or 2 carriage return/linefeeds (cr/lfs) before each row

/m2 or /mb2 of a matrix with more than one row.

/m3 or /mb3 print "Row 1", "Row 2"... before each row of a matrix with more than one row.

/ma1, /ma2 print 1 or 2 cr/lfs after each row of a matrix with more than one row.

/b1, /b2 print 1 or 2 cr/lfs before each row of any matrix.

/b3 print "Row1", "Row 2"... before each row of any matrix.

/a1, /a2 print 1 or 2 cr/lfs after each row of any matrix.

If nothing is specified **/m0** is assumed.

/jnt options: **These are important.**

/rd right-justified, [-]######.###. prec = Number of digits after "."

/re right-justified, [-]#.###E+-##. prec = Number of digits after "."

/ro /rd or **/re** format, depending on which is more compact. **/re** used only if exponent is less than -4 or greater than precision. If **/re** used, "." always appears.

/rz same as **/ro**, except that if **/re** used, trailing zeros are suppressed, and "." appears only if digits follow it.

/ld, /le, /lo, /lz are like **/rd, /re, /ro, /rz**, but left-justified.

To specify trailing character, follow /rd..../lz with S for space, C for comma, T for tab character, N for no trailing character. Default = space. This is important for outputting data you want to transfer to EXCEL.

fld specifies the minimum field width for printing a number**. fld** will be overridden if a number is too big to fit in the field specified.

For numeric values in matrices, **prec** sets the number of significant digits to be printed.

The default formats are:

Matrices	**format /mat /on /mb1 /ros 16,8;**
String arrays	**format /sa /on /mb1 /ros 16,-1;**
Strings	**format /str /off /mb1 /ros 16,-1;**

A single number is considered to be a 1×1 matrix. A **format** statement remains in effect until it is overridden by a new **format** statement. Complex numbers are printed with the sign of the imaginary half separating them and a trailing i to the imaginary half. The **[fld,prec]** parameter applies to **each** half of the components of a complex number. The total width of a field will be overridden if the number is too large to fit, e.g., **format /rds 1,0** can be used to print integers with a single space between them regardless of the size of the integers.

Examples of format statements and their consequences are in **basic02.gss**, reproduced in the appendix to these notes.

d. An Example of Data Input and Output Using a Comma Delimited File, **LAWLEY.CVS.**

See **basic03.gss,** reproduced in the appendix.

3. STRINGS AND STRING CONCATENATION

Although the GAUSS manual has a great deal to say about strings, I will deal with them here only minimally. Strings are sequences of characters. They are created by assigning a variable in GAUSS to have the value which is a sequence of characters enclosed in double quotes, e.g.,

```
x = "This program creates a string ";
```

And, of course, in the above examples you have already encountered print statements which print the string as output. String arrays are matrices with elements which are strings. String arrays may be created from string variables with the two string concatenation operators: **$|**, vertical concatenation, and **$~** horizontal concatenation. In contrast, the operator **$+**, adds two strings to form one long string, e.g.,

```
y = "to illustrate the concept.";

x $+ y = "This program creates a string to illustrate the concept.";
```

Elements in string arrays can be referenced in exactly the same way as numeric entries in a matrix.

A very important thing to remember about strings is that the backslash is used as an "escape" character inside double quotes in order to enter special characters such as tabs or spaces; therefore, the special character backslash is designated as \\ within a string. Hence, when entering DOS path names as string variables use \\ when you mean \; e.g.,

```
file = "c\\files\\gaussfls\\data\\input01";
```

Forward and back slashes appear to be equivalent for GAUSS inside strings.

4. NUMBERS AND MATRICES

a. Real and Complex Numbers

The following commands are available for controlling the precision or rounding numbers:

base10 Convert number to #.### and a power of 10.
Not implemented for complex numbers.
Format: { mant, pow } = base10(x);
Input: x scalar, number to break down.
Output: mant scalar, number in the range -10 < x < 10.
pow scalar, integer power such that:
mant * 10^pow == x

ceil Round up towards +INF.
Format: y = ceil(x);
Input: x NxK matrix
Output: y NxK matrix
This rounds every element in the matrix x to an integer. The elements are rounded up toward positive infinity.

floor Round down towards -INF.
Format: y = floor(x);
Input: x NxK matrix.
Output: y NxK matrix containing the elements of x rounded down.

This rounds every element in the matrix x "down" to the nearest integer.

prcsn Set computational precision for some matrix operations. You should not use this option. All calculations in GAUSS are done in double precision, with the exception of some of the intrinsic functions, which may use extended precision (18-19 digits of accuracy).

round Round to the nearest integer.
Format: y = round(x);
Input x NxK matrix.
Output y NxK matrix containing the rounded elements of x.

trunc Truncate decimal portion of number.
Format: y = trunc(x);
Input: x NxK matrix.
Output: y NxK matrix containing the truncated elements of x.

round, trunc, ceil and **floor** convert floating point numbers into integers. The internal representation for the converted integer is still 64-bit (double precision).

Most operators and functions operate on complex scalars and matrices in the expected way with no extra programming necessary but there are some special commands for complex numbers:.

complex Converts 2 real matrices to 1 complex matrix.
Format: y = complex(xr,xi);
Input: xr scalar or NxK real matrix containing the real part.
xi scalar or NxK real matrix containing the imaginary part.
Output: y NxK complex matrix.

real Returns real part of complex matrix.
Format: y = real(x);
Input: x NxK complex matrix.
Output: y NxK real matrix containing the real part of x.

imag Returns imaginary part of complex matrix. Same format as real.

iscplx Returns 1 (TRUE) if its argument is complex, 0 otherwise.
Format: y = iscplx(x);
Input: x NxK matrix
Output: y scalar, 1 if the input matrix is complex, 0 otherwise .

This does not check the imaginary component to see if it contains all zeros. Compare this with **hasimag,** which does check the imaginary part of a matrix.

These commands are illustrated in **basic04.gss**.

b. More on Creating Matrices: Special Matrices and Concatenation

The following commands are available for creating matrices and vectors:

design Creates a design matrix of 0's and 1's from a column vector of numbers specifying the columns in which the 1's should be placed.

Format: y = design(x);
Input: x Nx1 vector.
Output: y NxK matrix, where K = maxc(x); each row of y will contain a single 1, and the rest 0's. The one in the ith row will be in the round(x[i,1]) column.
Note that x does not have to contain integers. Each element will be rounded to nearest in any case.

eye Creates an identity matrix.

Format: y = eye(n);
Input: n scalar, the size of identity matrix to be created.
Output: y NxN identity matrix.
n is truncated to an integer if necessary. The matrix created contains 1's on the main diagonal and 0's everywhere else.

let Creates a matrix from a list of values.
We had this before:

Format: let x = { const_list };
x will have as many elements as are listed in const_list. Elements within a row are separated by spaces, rows are separated by commas. List the elements in row major order.

ones Creates a matrix of ones.

Format: y = ones(r,c);
Input: r scalar, number of rows.
c scalar, number of columns.
Output: y RxC matrix of ones.
r and c are truncated to integers if necessary.

recserar Computes auto-regressive recursive series.

Format: y = recserar(x,y0,a);
Input: x NxK matrix
y0 PxK matrix
a PxK matrix
Output: y NxK matrix containing the series.
recserar is particularly useful in dealing with time series. Typically, the result is thought of as K vectors of length N. y0 contains the first P values of each of these vectors (thus, these are prespecified). The remaining elements are constructed by computing a Pth order "autoregressive" recursion, with weights given by a, and then by adding the result to the corresponding elements of x. That is, the tth row of y is given by:

$$y(t) = x(t) + a(1)y(t-1) + \ldots + a(P)y(t-P), \quad t = P+1,\ldots,N$$

and

$$y(t) = y0(t), \quad t = 1,\ldots,P$$

Note that the first P rows of x are not used.

recsercp Computes recursive series involving products. Can be used to compute cumulative products, to evaluate polynomials using Horner's rule, and to convert from base b representations of numbers to decimal representations, among other things.

Format: y = recsercp(x,z);
Input: x NxK or 1xK matrix.
z NxK or 1xK matrix.
Output: y NxK matrix in which each column is a series generated by a

recursion of the form:

$y(1) = x(1) + z(1)$

$y(t) = y(t-1)*x(t) + z(t),\ t = 2,...,N$

recserrc Computes recursive series involving division. Can be used, among other things, to convert from decimal to other number systems (radix conversion).

Format: y = recserrc(x,z);

Inputs: x 1xK or Kx1 matrix.

z NxK matrix (the columns of z must equal the number of elements in x).

Output: y NxK matrix in which each column is a series generated by a recursion of the form:

$y(1) = x \bmod z(1)$

$y(2) = \text{trunc}(x/z(1)) \bmod z(2)$

$y(3) = \text{trunc}(x/z(2)) \bmod z(3)$

...

$y(n) = \text{trunc}(x/z(n-1) \bmod z(n)$

seqa Creates a vector as an additive sequence.

Format: y = seqa(start,inc,n);

Input: start scalar specifying the first element.

inc scalar specifying the increment.

n scalar specifying the number of elements in the sequence.

Output: y Nx1 vector containing the specified sequence.

y will contain the first element equal to start, the second equal to start+inc, and the last equal to start+inc*(n-1). For instance, seqa(1,1,10) creates a column vector containing the numbers 1,2...10.

seqm Creates a vector as a multiplicative sequence.

Format: y = seqm(start,inc,n);

Input: start scalar specifying the first element.

inc scalar specifying the increment.

n scalar specifying the number of elements in the sequence.

Output: y Nx1 vector containing the specified sequence.

y will contain the first element equal to start, the second equal to start*inc, and the last equal to start*inc^(n-1). For instance, seqm(10,10,10) creates a column vector containing the numbers 10, 100...10^10.

toeplitz Creates a Toeplitz matrix from a column vector.[1]

---

[1] For a discussion of the importance of Toeplitz matrices in computational mathematics and statistics see Press, W. H., B. P. Fannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes: the Art of Scientific Computing.* New York: Cambridge University Press, 1986, pp.47 - 52. An n by n Toeplitz matrix is specified by giving 2n - 1 numbers R_k, k = -n+1, ..., -1, 0, 1, ..., n-1, which are then arranged in the form of a matrix R having constant elements along the diagonals of R, e.g.,

$$R = \begin{Bmatrix} R_0 & R_{-1} & \dots & R_{-n+2} & R_{-n+1} \\ R_1 & R_0 & \dots & R_{-n+3} & R_{-n+2} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ R_{n-2} & R_{n-3} & \dots & R_0 & R_{-1} \\ R_{n-1} & R_{n-2} & \dots & R_1 & R_0 \end{Bmatrix}.$$

Imagine solving the system of equations Rx = y, i..e.,

Format: t = toeplitz(x);
Input: x Kx1 vector.
Output: t KxK Toeplitz matrix.

zeros Creates a matrix of zeros.
Format: y = zeros(r,c);
Input: r scalar, the row dimension.
c scalar, the column dimension.
Output: y RxC matrix containing only zeros.
r and c are truncated to integers if necessary.

| Vertical concatenation operator.

~ Horizontal concatenation operator.

zeros or **ones** can be used to create a constant vector or matrix. The random number generators to be discussed below also can be used to create matrices, which, however, vary from run to run. I will frequently use **rndn** or **rndu** to create matrices to illustrate matrix operations. You can use the **CEIL** command to create a sequence of random integers: **ceil(rndu(rows, columns)) * 100);** will produce a matrix rows by columns of random integers.

These commands are illustrated in **basic05.gss**.

c. Matrix Description and Manipulation

(1) Dimensions

cols(x) Returns number of columns in the matrix x.

rows(x) Returns number of rows in the matrix x.
If x is an empty matrix, rows and cols return 0..

(2) Minimum and Maximum Elements, Ranges and Counts, Means, Sums and Products

maxc(x) Returns largest element in each column of the matrix x.
To find the maximum maximorum, apply maxc twice as **maxc(maxc(x)').**

maxindc(x) Returns row number of largest element in each column of the matrix x.

minc(x) Returns smallest element in each column of the matrix x.

minindc(x) Returns row number of smallest element in each column of the matrix x.

sumc(x) Computes the sum of each column of the matrix x.

cumsumc(x) Computes cumulative sums of each column of the matrix x.

prodc(x) Computes the product of each column of the matrix x.

cumprodc(x) Computes cumulative products of each column the matrix x.

meanc(x) Computes mean value of every column of the matrix x.

stdc (x) Computes standard deviation of every column of the matrix x.
stdc essentially computes:

sqrt(1/(N-1)*sumc((x-meanc(x)')^2))

Thus, the divisor is N-1 rather than N, where N is the number of elements being summed. To convert to the alternative definition, multiply by
sqrt((N-1)/N).

$$\sum_{j=1}^{n} R_{i-j} x_j = y_i \text{ , i =1,.., n,}$$

for the x_j's, j = 1, ..., n. This formulation makes clear the importance of Toeplitz matrices in all kinds of recursive calculations.

sqrt and ' are discussed below.

An important general command for generating a variety of descriptive statistics is **dstat(0,x).**

Format: **{ vnam,mean,var,std,min,max,valid,mis } = dstat(dataset,vars);**
Input: dataset If **dataset** is 0, **vars** will be assumed to be a matrix containing the data.
vars If **dataset** is 0, then **vars** will be interpreted as an NxM matrix of the data to be analyzed.
Output: vnames Kx1 character vector, the names of the variables used in the statistics. By default, the variables names will be X1,X2,..Xm
mean Kx1 vector, means.
var Kx1 vector, variance.
std Kx1 vector, standard deviation.
min Kx1 vector, minima.
max Kx1 vector, maxima.
valid Kx1 vector, the number of valid cases.
missing Kx1 vector, the number of missing cases. GAUSS's missing value code is ".".
I will deal with the options for **dstat** in more detail at a later point in this tutorial.

counts(x,v) Returns number of elements of a vector falling in specified ranges.
Input: x Nx1 vector containing the numbers to be counted.
v Px1 vector containing break-points specifying the ranges within which counts are to be made. The vector v **must** be sorted in ascending order.

Output: c Px1 vector, the counts of the elements of x that fall into the regions:
x <= v[1]
v[1] < x <= v[2]
.
.
v[p-1] < x <= v[p]
If the maximum value of x is greater than the last element (the maximum value) of v, the sum of the elements of the result, c, will be less than N, the total number of elements in x.

indexcat(x,v) Returns indices of elements falling within specified ranges.
Inputs: x Nx1 vector.
v scalar or 2x1 vector.
If scalar, the function returns the indices of all elements of x equal to v,
If 2x1, then the function returns the indexes of all elements of x that fall into the range:
v[1] < x <= v[2].
If v is scalar it can contain a single missing value symbol "." to specify the missing value as the category.
Output: y Lx1 vector, containing the indexes of the elements of x which fall into the category defined by v. It will contain error code 13 if there are no elements in this category.
A loop, to br discussed below, can be used to pull out indices of multiple categories.

rankindx(x,flag) Returns rank index of Nx1 vector.
Inout x Nx1 vector.

flag scalar, 1 for numeric data or 0 for character data.

Output: y Nx1 vector containing the ranks of x. The rank of the largest element of x is N, and the rank of the smallest element of x is 1. To get ranks in descending order, subtract y from N+1. rankindx assigns different ranks to elements that have equal values (ties). Missing values are assigned the lowest ranks.

(3) Matrix Element Manipulation

rev(x) Reverses the order of rows of the matrixx.

rotater(x, r) Rotates the rows of the matrix x by r elements, wrapping elements. The rotation is performed horizontally within each row of the matrix. A positive rotation value moves elements to the right. A negative rotation value moves elements to the left. The elements pushed off the end of the row wrap around to the opposite end of the same row. If the rotation value is greater than or equal to the number of columns in the matrix x, it is recalculated as (r mod cols(x)).

shiftr(x,s,f) Shifts rows of the matrix x, filling in holes with a specified value.

Input: x NxK matrix to be shifted.
s Nx1 or 1x1 matrix specifying the amount of shift.
f Nx1 or 1x1 matrix specifying the value with which to fill in vacated elements.

The shift is performed horizontally within each row of the matrix. A positive shift value moves elements to the right. A negative shift value moves elements to the left. The elements pushed off the end of the row are lost, and the fill value is used for the new elements on the other end. If the shift value is greater than or equal to the number of columns in the matrix x, it is recalculated as (s mod cols(x)).

reshape(x,r,c) Reshapes a matrix to new dimension.

Input: x NxK matrix, string array, or string.
r scalar, new row dimension.
c scalar, new column dimension.

Output: y RxC matrix or string array created from the elements of x.

Matrices and string arrays are stored in **row major order**, that is, the first c elements of x are put into the first row of y, the second into the second row, and so on. If there are more elements in x than in y, the remaining elements are discarded. If there are not enough elements in x to fill y, when reshape runs out of elements it goes back to the first element of x and starts getting additional elements from there.

vec Stacks columns of a matrix to form a single column.

vecr Stacks rows of a matrix to form a single column.

X.' Bookkeeping transpose of matrix X. The standard transpose operator (') returns the complex conjugate transpose of complex matrices. The bookkeeping transpose (.') simply transposes the matrix without changing the sign of the imaginary part.

The functions **reshape, vec, vecr** and the dot transpose operator **(.')** change the shape of matrices, while **rev, rotater and shiftr** move elements in the matrix, but retain the structure of the matrix.

(4) Sorting Data

sortc(x,c) Sorts rows of matrix based on numeric column.
Input: x NxK matrix.
c scalar specifying one column of x to sort on.

Output: y NxK matrix equal to x and sorted on the column c.

sortc sorts the rows of a matrix with respect to a specified column. That is, it sorts the elements of a column and arranges the rows of the matrix in the same order as the sorted column.
The matrix may contain both character and numeric data, but the sort column must all be of one type. sortc assumes that the column to sort on contains numeric data. Missing values sort below everything else.
If you need to obtain the matrix sorted in descending order, you can use: **rev(sortc(x,c));**

sortind(x) Returns a sorted index of numeric vector.

unique(x, flag) Sorts and removes duplicate elements of a vector.
Input: x Nx1 or 1xN vector.
flag scalar, 1 for numeric data or 0 for character data.
Output: y Mx1 vector, sorted x with duplicates removed.
x is sorted in ascending order. Missing values sort below everything else.

uniqindx(x,flag) Returns a sorted unique index of vector, i.e. the indices corresponding to the sorted values remaining after application of **unique**.

Sorting routines operate on matrices by sorting the rows on the basis of a key column. Both character and numeric data can be sorted using these functions.

(5) Submatrix Extraction

delif(x,e) Deletes rows from a matrix using a logical expression.
Input: x NxK data matrix.
e Nx1 logical vector (vector of 0's and 1's).
Output: y MxK data matrix consisting of the rows of y for which there is a 0 in the corresponding row of e; y will be a scalar missing "." if no rows remain.
The input e will usually be generated by a logical expression.

selif (x,e) Similar to **delif** . Selects rows from a matrix. Those selected are those for which there is a 1 in the corresponding row of e.

packr(x) Deletes the rows of a matrix that contain any missing values, ".".
Input: x NxK matrix.
Output: y LxK submatrix of x containing only those rows that do not have missing values in any of their elements.
The GAUSS code for a missing value is ".". The best procedure is to replace whatever other code is used in your data with the GAUSS code <u>before</u> you import your data, although GAUSS has an internal procedure for conversion.

diag(x) Creates a column vector from the diagonal of a matrix.
Input: x NxK matrix.
Output: y min(N,K) by 1 vector.
The matrix x need not be square.

diagrv(x,v) Inserts a vector into the diagonal of a matrix.
Input: x NxK matrix.
v min(N,K) vector.
Output: y NxK matrix equal to x with its principal diagonal elements equal to those of v.

Note that **diag** reverses the procedure and pulls the diagonal out of a matrix.

lowmat(x) Returns the lower portion of a matrix. **lowmat** returns the main diagonal and every element below. **lowmat1** is the same except it replaces the main diagonal with ones.

Input: x NxK matrix

Output: L NxK matrix containing the lower elements of the matrix. The upper elements are replaced with zeros.

upmat(x) Returns the lower portion of a matrix. **upmat** returns the main diagonal and every element above. **upmat1** is the same except it replaces the main diagonal with ones. The upper elements are replaced

submat(x,r,c) Extracts a submatrix of a matrix, with the appropriate rows and columns given by the elements of matrices.

Input: x NxK matrix
r LxM matrix of row indices
c PxQ matrix of column indices

Output: y (L*M) by (P*Q) submatrix of x. y may be larger than x. If r = 0, all rows of x are extracted. If c = 0, all columns of x are extracted.

trimr(x,t,b) Trims rows from the top and/or bottom of a matrix.

Input: x NxK matrix from which rows are to be trimmed.
t scalar, the number of rows which are to be removed from the top of the matrix x.
b scalar, the number of rows which are to be removed from the bottom of the matrix x.

Output: y RxK matrix, where R=N-(t+b), containing the rows left after the trim.

If either t or b is zero, no rows are trimmed from that end of the matrix.

Elements of a matrix and submatrices can also be extracted using the row and column indices as follows:

x[r1:r2,c1:c2]

extracts the submatrix consisting of rows r1 to r2 and columns c1 to c2. Using a single index for column and row obviously extracts just the rc-th element. **Using a dot, ".", in place of an index, extracts all the row or the column elements, as the case may be.** Thus **x[.,.]** is just x – all of it.

Examples of the above GAUSS commands are given in **basic06.gss**.

5. MATHEMATICAL AND STRING OPERATORS

The mathematical, matrix, string array, and string operators are:

MATHEMATICAL OPERATORS

+ addition
- subtraction
*** multiplication**
.* Element by Element multiplication
^ Element by Element exponentiation
! factorial Nonintegers are rounded to the nearest integer before the factorial operation is applied. Cannot be applied to a negative number.
./ Element by Element division
/ division or linear equation solution of Ax = b, for example: x = b/A;

% **modulo division**
.*. **Kronecker product**
***~** **horizontal direct product** Matrices must have the same number of rows; result has cols. = product of cols.

Matrix division, /, is more complex than other operations in GAUSS: In the expression x = b/A, if A and b are scalars $A \neq 0$, the operator is ordinary scalar division; if b and A are conformable, the operator / solves the system of linear equations Ax = b as follows:

If A is square and has the same number of rows as b, then b/A will solve the system of linear equations Ax = b using the LU decomposition. Thus A need not be of full rank.

If A is rectangular with the same number of rows as b, then b/A yields the OLS solution to A'Ax = A'b using the Cholsky decomposition.

These interpretations stand in contrast to the use of the GAUSS operators INV and INVPD in the statements x = inv(A'A)*A'b and x = invpd(A'A)*A'b, but produce results similar to INVSWP. The relation to the GAUSS procedures OLS and OLSQR is discussed below.

LU decomposition: Computes the lu decomposition of a square matrix with partial (row) pivoting.
Format: { L,U } = lu(x);
Input: x NxN nonsingular matrix.
Output: L NxN "scrambled" lower triangular matrix. This is a lower triangular matrix that has been reordered based on the row pivoting.
U NxN upper triangular matrix.
This performs an LU decomposition of a matrix X such that:

L * U = X.

Thus x = b/A is equivalent to **{ L,U } = lu(A);** followed by **x = lusol(b,L,U);**

We have already dealt with the following operators:

MATRIX ARRAY OPERATORS:
| **matrix vertical concatenation**
~ **horizontal concatenation**
' **transpose**
.' **bookkeeping transpose** Primarily for handling complex matrices.
STRING OPERATORS:
$+ **string addition**
$| **string array vertical concatenation**
$~ **string array horizontal concatenation**

A reminder: Symbols used for indexing matrices are: "[", "]", "." and ":". For example,

x[1 2 5] returns the 1st, 2nd and 5th elements of x.
x[2:10] returns the 2nd through 10th elements of x.
x[.,2 4 6] returns all rows of the 2nd, 4th, and 6th columns of x.

The mathematical operators are illustrated in **basic07.gss.**

6. RELATIONAL AND LOGICAL OPERATORS FOR SCALARS OR MATRICES

The relational operators which return a scalar 0 or 1 are:

Less than:	**Not equal:**
z = x < y;	z = x /= y;
z = x lt y;	z = x ne y;
Greater than:	**Greater than or equal to:**
z = x > y;	z = x >= y;
z = x gt y;	z = x ge y;
Equal to:	**Less than or equal to:**
z = x == y;	z = x <= y;
z = x eq y;	z = x le y;

The result is a scalar 1 (TRUE) or 0 (FALSE), based upon a comparison of all elements of x and y. ALL comparisons must be true for a result of 1 (TRUE). It is a matter of indifference whether you use the alphabetic or the symbolic form of the operator; however, in the alphabetic form the operator must be preceded and followed by a space.

If you want to compare strings or character data, use the symbolic form preceded by $, e.g., $==. The meaning of an equality comparison is obvious. For comparisons other than equality, GAUSS does a bit-by-bit comparison, so you have to know how the characters are coded.

If you are dealing with conformable matrices and want an element by element comparison which will return a **matrix** of zeros and ones, precede the operator in symbolic or alphabetic form by a dot ".", e.g., .== or .eq. **Always leave spaces around dot operators,** otherwise GAUSS may interpret the dot as a decimal point.

The logical operators perform logical or Boolean operations on numeric values. On input a nonzero value is considered TRUE and a zero value is considered *FALSE.* The logical operators return a 1 *if TRUE* and a 0 *if FALSE.* The following operators require scalar arguments. These are the ones to use in **if** and **do** statements.

Complement	z = not x;
Conjunction	z = x and y;
Disjunction	z = x or y;
Exclusive or	z = x xor y;
Equivalence	z = x eqv y;

Decisions are based on the following truth tables.

Complement

X	not X
T	F
F	T

Conjunction

X	Y	X and Y
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction

X	Y	X or Y
T	T	T
T	F	T
F	T	T
F	F	F

Exclusive Or

X	Y	X xor Y
T	T	F
T	F	T
F	T	T
F	F	F

Equivalence

X	Y	X eqv Y
T	T	T
T	F	F
F	T	F
F	F	T

If the logical operator is preceded by a dot "." the result will be a matrix of 1's and 0's based upon an element-by-element logical comparison of x and y. For complex matrices, the logical operators consider only the real part of the matrices. These operators are useful for editing data but not in do loops or if statements.

Basic08.gss illustrates these operators.

7. PROGRAM FLOW CONTROL: DO-LOOPS, FOR-LOOPS, AND BRANCHING

Looping is controlled with the **do** statement.

```
do while scalar_expression;      /* loop if true */
.
.
.
endo;

do until scalar_expression;      /* loop if false */
  .
  .
  .
endo;
```

Inside a do-loop

break; causes a jump to the statement following **endo**.

continue; causes a jump to the top of a do loop.

Do-loops often make use of counters, which have to be initialized before the start of the loop and augmented during the loop. A **for**-loop is equivalent to this kind of a do-loop, but handles initialization and augmentation of the counter automatically.

```
for i (start, stop, step);
        .
        .
endfor;
```

Input: i the name of the counter variable.
start scalar expression, the initial value of the counter.
stop scalar expression, the final value of the counter.
step scalar expression, the increment value.

These arguments are strictly local to the loop. The expressions, start, stop and step are evaluated only once when the loop initializes. They are converted to integers and stored local to the loop. **This means you cannot use i to index the elements of a matrix or a vector you will later retrieve. Instead, use a**

different counter and set to the index for the elements of the matrix internal to the loop. A for-loop is optimized for speed and much faster than a do-loop. The commands **break** and **continue** are supported. The **continue** command steps the counter and jumps to the top of the loop. The **break** command terminates the current loop by jumping past the **endfor** statement. When the loop terminates, the value of the counter is stop if the loop terminated naturally. If break is used to terminate the loop and you want the final value of the counter you nee to assign it to a variable before the break statement.

Conditional branching is done with the if statement:

```
if scalar_expression;
    .
    .
elseif scalar_expression;
    .
    .
else;
    .
    .
endif;
```

The expression after the if or the **elseif** must be an expression which returns a scalar. Use the relational and logical operators without the dot. **elseif** and **else** are optional. There can be multiple **elseif**'s.

Do-loops, for-loops and if statements can all be nested.

Unconditional branching is done with **goto**. The target of a goto is called a label. Labels must begin with '_' or an alphabetic character and **are always followed by a colon**.

Since relational operators return 0's or 1's and since dot relational operators return vectors of 0's and 1's, they can be used to accomplish many of the things done by do-loops or for-loops in a more efficient manner.

Do-loops, for-loops and branching are illustrated in basic09 as well as the use of dot relational operators to accomplish the same thing. **Goto**, **break** and **continue** statements and **labels** are illustrated in **basic09.gss.**

8. PROCEDURES, FUNCTIONS, KEYWORDS AND LIBRARIES

A procedure consists of a body of GAUSS code surrounded by four GAUSS commands:

proc	Begin definition of multiline procedure.
local	Declare variables local to a procedure.
	Body of GAUSS procedure.
retp	Returns from a procedure.
endp	Terminate a procedure definition.

A procedure always begins with a **proc** statement and always ends with an **endp** statement. Between these is the body of the procedure itself. The **local** and the **retp** serve to indicate where GAUSS is to "look for" the variables it uses in the procedure. There are three ways of passing data to a GAUSS procedure: (1) through arguments of the procedure; (2) implicitly as a variable which is defined in the main body of the program which calls the procedure; and (3) creating the data internally within the procedure, in which case it is given a name which is strictly local to the procedure. Neither the name nor the data to which the name refers have any existence outside the procedure. These three types of variables, defined in relation to a procedure, are called ARGUMENTS, GLOBAL VARIABLES and LOCAL VARIABLES, respectively.

Local variables are not really necessary for the operation of a procedure, which could, conceivably, operate entirely on variables defined elsewhere in the program. The forms of these statements are listed below. I have put @ sign around parts which are optional and may be omitted.

proc @(number of returns) =@
NAME(argument list, @&names of procedures called by this procedure@);

If no number of returns are listed, GAUSS will assume that their is only one or, perhaps, none; i.e., that the procedure changes some values of global variables. The names of procedures which may be called by this one must be preceded by an ampersand. Both variables and procedure names are "stand-ins" for the names you will fill in when you call the procedure.

local variable names separated by commas, @names of procedures:**proc**@;

The statement above would place the variable names and the procedure names in the local symbol table for the current procedure being compiled. A **local** statement is legal only between the **proc** statement and the **endp** statement of the procedure definition. These symbols cannot be accessed outside of the procedure. Multiple local statements are allowed. The symbol after the ampersand in the list of arguments will be treated as a procedure whenever it is accessed in the procedure. What is actually passed in is a pointer to a procedure. Even if you use a procedure defined elsewhere in your program you need not include it in the list of arguments or in a local declaration as long as you use the globally defined name of the procedure in the first procedure. What GAUSS does if you include the name of a procedure in the list of arguments is to fill it in the defined procedure in place of the one locally named. For example, as you may know, many GAUSS commands are in fact procedures; some such as **gradp(&f, parameters)** call other procedures (in this case, a procedure which defines a single valued function and returns a vector of gradients of the function you fill in as the argument with respect to its arguments at a point you have specified in the vector parameters). Naturally, the supplied procedures on which the command operates change from call to call.

Sometimes it's necessary to use some data in computing the function you're going to fill in.In **gradp**, for example, f is a likelihood function; since **gradp** doesn't allow anything but the single valued function and a list of parameters to appear as arguments, the data required by f must be treated as globally defined variables. But, in general, it is less dangerous to pass information to a procedure through a list of arguments and to treat all variables on which the procedure operates as local. Otherwise the procedure might change some of the variables *outside* your procedure without you're knowing about it. Likewise, although the actual names of arguments and local variables in a procedure definition are "stand-ins," it is better not to use names which are the same as those of global variables.

retp@(x)@;

0-31 items may be returned from a procedure in a retp statement. The items may be expressions. Items are separated by commas.It is legal to return with no arguments, as long as the procedure is defined to return zero arguments.

endp;

marks the end of a procedure definition that began with the **proc** statement.

How you call a procedure in a program depends on whether it has any returns and on the number of such returns: For a procedure with no returns, a simple statement of the name of a procedure with its arguments filled in will do; or you can say **call procedure(arguments)**, in which case all returns, if any, will be ignored. In this case, however, you may have trouble getting at the returns. For procedure with a single return, you can get it by a simple assignment statement, e.g., **y = procedure(aruments)**, will simply assign the return to the variable y. Of course, this variable could be a vector or a matrix rather than a scalar. Alternatively, for procedures with multiple returns, use **{ return1, return2, ... } = procedure(arguments);.** The number of items listed on the left must be the same as the number in the **retp(...);** statement, which, in turn must equal the number in the **proc(....)** statement defining the procedure.

All of these matters are illustrate in **basic10.gss**.

Functions are one-line procedures which have a single output. They are created with the GAUSS command **fn:**

Format: **fn name**(arguments) = code for function;

Functions can be called in the same way as other procedures which return a single value. I think it's cleaner to use procedures.

Keywords are something else again. They are rarely used, except by programmers writing packages. Essentially they are procedures which take a string as an argument.

GAUSS allows for a way to collect frequently used procedures in a "library" which will be accessed whenever the procedure name is referenced in a program you are writing. In fact, many of GAUSS's basic functions are procedures collected in this way in the group of files with .src tags collected in a folder called src in the GAUSS directory which contains the GAUSS.exe which runs GAUSS. When you call such a procedure in your program, this is where GAUSS goes to get it. There are also groups of related procedures collected in special libraries, such as, for example, the graphing procedures taken up in the next section. These are not automatically accessed unless you make them active by a statement in your program, e.g., **library pgraph; graphset;** activates the graphics library and resets all of the global variables controlling these procedures to their default values. You can do much the same thing with a group of your own procedures. However, doing this sort of thing makes your programs less accessible to others. My recommendation is to use the standard GAUSS libraries and to collect your own procedures in one place at the end of your programs. That way your programs will be portable.

9. SIMPLE GRAPHS IN GAUSS

There is nothing more helpful in analyzing data than visualization. GAUSS offers a variety of graphical procedures which are accessed through the pgraph library. Near the beginning of any program in which you intend to do any graphing, include the line

library pgraph; graphset;

GAUSS provides a complete windowing system for plotting multiple graphs on one page. You can print the graphs that GAUSS generates directly, but if you want to put them into a WORD document or manipulate them outside of GAUSS you will have to convert them to another format GAUSS supports conversion to four formats: PostScript, Lotus .PIC, HPGL Plotter and PCX Bitmap. unfortunately WORD 7.0 will no longer accept any of these types. The best for conversion and import into a WORD document used to be HPGL (*.hgl) files. If you have access to WORD 6.0 or lower you can still insert *.hgl files into a WORD document; otherwise you'll have to get a separate package of filters: Try INSO's *ImageStream97*, http://www.inso.com

The following graph types are supported by GAUSS. The procedure names are listed on the left; I will demonstrate only **xy**, **surface**, **contour**, and **bar**.

xy Graph X,Y using cartesian coordinate system.
Format: xy(x,y);
Inputs: x NxM or Nx1 matrix. Each column represents the x values for a particular line.
y NxM or Nx1 matrix. Each column represents the y values for a particular line.

logx Graph X,Y using logarithmic X axis

logy Graph X,Y using logarithmic Y axis

loglog Graph X,Y using logarithmic X and Y axes

bar Generate bar graph.
Format: bar(val,ht);

Inputs: val Nx1 numeric vector. If scalar 0, a sequence from 1 to rows(ht) will be created.
ht NxK numeric vector, bar heights. K sets of bars will be graphed.

_pbarwid Global scalar, width of bars. The valid range is 0-1. If this is 0 the bars will be a single pixel wide, if 1 the bars will touch each other.
If this value is positive, the bars will be overlapping and care on the part of the user must be taken to set the bar shades accordingly.
If this value is negative, the bars will be plotted side-by-side. There is no limit to how many bars may be plotted. The default value for **_pbarwid** is .5 (overlapping).

_pbartyp Global 1x2 or Kx2 matrix. The first column controls the shading, the second column controls the color. Each row of shade and color corresponds to a column of bar data. Shade types are:

0 no shading
1 dots
2 horizontal lines
3 diagonal lines, positive slope
4 diagonal lines, negative slope
5 diagonal crosshatch
6 solid

hist	Compute and graph frequency histogram
histf	Graph a histogram given a vector of frequency counts
histp	Graph a percent frequency histogram of a vector
box	Graph data using the box graph percentile method
xyz	Graph X, Y, Z using 3D cartesian coordinate system
surface	Graph a 3D surface.

Format: surface(x,y,z);
Inputs: x 1xK vector, the min and max values for the X axis.
y Nx1 vector, the min and max values for the Y axis.
z NxK matrix, heights of the mesh above the X-Y plane.

contour Graph contour data.

Format: contour(x,y,z);
Inputs: x 1xK vector, the min and max values for the X axis.
y Nx1 vector, the min and max values for the Y axis.
z NxK matrix, heights above the X-Y plane.

draw Supply additional graphic elements to graphs

The following global variables and procedures are used to control the properties of your graphs. All of them are listed below by category. The ones I will demonstrate are bold faced and described:

Axes Control and Scaling

_paxes Turn axes on or off. When you overlay two graphs you will want to reset this.
A scalar, 2x1 or 3x1 vector for independent control of each axis. If a scalar value, the vector will be expanded that value.

```
Examples: _paxes = 0;      - turn off all axes
          _paxes = 1;      - turn on all axes
          _paxes = {1,0}   - turn X axis, Y off;
          _paxes = {0,0,1} - turn Z axis on, X,Y off;
```

_pcrop	Control cropping of graphics data outside axes area
_pcross	Controls where axes intersect
_pframe	Draw a frame around 2D, 3D plots.

A 2 x1 vector: {1,1} = frame on with tick marks, default; {0,0} = frame off, no tick marks; etc.

_pgrid Control major and minor grid lines

_pxpmax Control precision of numbers on X axis
_pypmax Control precision of numbers on Y axis
_pzpmax Control precision of numbers on Z axis
_pxsci Control use of scientific notation on X axis
_pysci Control use of scientific notation on Y axis
_pzsci Control use of scientific notation on Z axis
_pticout Control direction of tic marks on axes

scale Scale X,Y axes on 2-D plots.

scale3d Scale X,Y,Z axes for 3-D plots.

These fix a scaling for subsequent graphs.

Format: scale(x,y);
scale3d(x,y,z);

Input: x matrix, the X axis data.
y matrix, the Y axis data.
z matrix, the Z axis data.

x, y, and z must each have at least 2 elements. Only the minimum and maximum values are necessary. This routine fixes the scaling for all subsequent graphs until graphset is called. This also clears **xtics**, **ytics** and **ztics** whenever it is called. See below.

If either of the arguments is a scalar missing, autoscaling will be restored for that axis.

If an argument has 2 elements, the first will be used for the minimum and the last will be used for the maximum. If an argument has 2 elements, and contains a missing value, that end of the axis will be autoscaled. If you want direct control over the axes endpoints a and tick marks use xtics, ytics, or ztics. If these have been called after scale, they will override scale.

xtics Scale X axis and control tic marks

ytics Scale Y axis and control tic marks

ztics Scale Z axis and control tic marks

These set and fix scaling, axes numbering and tics for the X axis, the Y axis and the Z axis, respectively. The format is the same.

Format: xtics(min,max,step,minordiv);

Input: min scalar, the minimum value.

max scalar, the maximum value.
step scalar, the value between major tics.
minordiv scalar, the number of minor subdivisions.

This routine fixes the scaling for all subsequent graphs until **graphset** is called and must be called after any call to **scale**, because scale resets **xtics** whenever it is called.

Text, Labels, Titles, and Fonts

_plegctl Sets location and size of plot legend, used in conjuntion with **_plegstr**.

[1] X,Y coordinate units: 1=plot coord, 2=inches, 3=Tek pixels
[2] Legend text font size. 1 <= size <= 9.
[3] X coordinate of lower left corner of legend box
[4] Y coordinate of lower left corner of legend box

Example: _plegctl = { 1 4 10 20 }

_plegstr Specify legend text entries. String, legend entry text. Text for multiple curves is separated by a null byte ('\000').

_paxht Control size of axes labels
_pnumht Control size of axes numbering
_pnum Axes number control and orientation
_pdate Date string contents and control
_ptitlht Control title size

xlabel X axis label.

ylabel Y axis label.

zlabel Z axis label

These set a label for the X axis, the Y axis and the Z axis, respectively. The format is the same.
Format: xlabel(str);
Input: str string, the label for the X axis.

title Specifies main title for graph.
Format: title(str);
Input: str string, the title to display above the graph.
str may contain up to 3 lines of title separated by line feed characters. On multiple line titles, the character size will be reduced automatically. This string may contain up to 180 characters total.

asclabel Define character labels for tic marks

fonts Load fonts for labels, titles, messages and legend

view Set 3D observer position in workbox units

viewxyz Set 3D observer position in plot coordinates

volume Sets length, width, and height ratios of 3D workbox

Main Curve Lines and Symbols

_pboxctl Control box plots

_plctrl Control main curve and data symbols.
< 0 draw symbols only every **_plctrl** points, no lines.
= 0 draw line only, no symbols.
> 0 draw line and symbols every **_plctrl** points.
Examples:
_plctrl = -1; symbol on all points, no curve drawn
_plctrl = -10; symbol every 10 points, no curve drawn
_plctrl = 1; symbol at all points, draw curves also
_plctrl = 10; symbol every 10 points, draw curves also

_pcolor Control line color for main curves. A scalar or a k x 1 vector, to set the colors for the main curves in xy, xyz, and log graphs. To use a single color for all curves set this equal to a color scalar value. Default values are given in **_psel** which you can find in pgraph.dec under the GAUSS src folder on the k-drive. See below for the color codes.

_pltype Control line style for main curves. Ascalar or k x 1 vector, line type for the main curves. If this is a nonzero scalar, all the lines will be of the same type. Default values are given in **_psel** which you can find in pgraph.dec under the GAUSS src folder on the k-drive.

_plwidth Control line thickness for main curves. May be zero or greater. 0 is the skiniest.

_pstype Control symbol type used at graph points. A scalar or a k x 1 vector. If a scalar, all symbols will be the same. See below for codes.

_psymsiz Control symbol size

_pzclr Z level color control for contour and 3D graphs. See **surface**.

Extra Messages, Symbols, Lines, Circles, Arrows, and Error Bars

_parrow Create arrows

_parrow3 Create arrows for 3d graphs

_pline Plot extra lines and circles in a 2D graph. Extra lines are valuable. I don't know when you'd want extra circles. An M x 9 matrix. Each line controls one item to be drawn. Default is nothing. Here are the specs:
[M,1] Item type and coordinate system
1 Line in plot coordinates
2 Line in inch coordinates
3 Line in pixel coordinates
4 Circle in plot coordinates
5 Circle in inch coordinates
6 Radius in plot coordinates
7 Radius in inch coordinates

[M,2] line type, 1-6. See below.
[M,3-7] coordinates and dimensions
(1) Line in plot coordinates.
[M,3] x starting point in plot coordinates
[M,4] y starting point in plot coordinates
[M,5] x ending point in plot coordinates
[M,6] y ending point in plot coordinates
[M,7] 0=continuation, 1=begin new curve
(2) Line in inch coordinates
[M,3] x starting point in inches
[M,4] y starting point in inches
[M,5] x ending point in inches
[M,6] y ending point in inches
[M,7] 0=continuation, 1=begin new curve
(3) Line in pixel coordinates
[M,3] x starting point in pixels
[M,4] y starting point in pixels
[M,5] x ending point in pixels
[M,6] y ending point in pixels
[M,7] 0=continuation, 1=begin new curve
(4) Circle in plot coordinates
[M,3] x center of circle in plot coordinates
[M,4] y center of circle in plot coordinates
[M,5] radius in plot coordinates
[M,6] starting point of arc in radians
[M,7] ending point of arc in radians
(5) Circle in inches
[M,3] x center of circle in inches
[M,4] y center of circle in inches
[M,5] radius in inches
[M,6] starting point of arc in radian
[M,7] ending point of arc in radians
(6) Radius in plot coordinates
[M,3] x center of circle in plot coordinates
[M,4] y center of circle in plot coordinates
[M,5] beginning point of radius in plot coord.
[M,6] ending point of radius
[M,7] angle in radians
(7) Radius in inches
[M,3] x center of circle in inches
[M,4] y center of circle in inches
[M,5] beginning point of radius in inches
[M,6] ending point of radius
[M,7] angle in radians
[M,8] color 1-15
[M,9] line thickness
Examples:
Draw a solid, red line in plot coordinates:
_pline = { 1 6 10 20 100 200 1 4 0 }
Draw a thick, solid, red line in inches:
_pline = { 2 6 1.0 2.0 3.0 4.5 1 4 5 }

_pline3d	Plot extra lines for 3d graphs
_psym	Plot extra symbols
_psym3d	Plot extra symbols for 3d graphs
_perrbar	Plot error bars

_pmsgctl Control position of message text. Prints extra messages. Message strings specified in **_pmgstr.**
[M,1] x location of message
[M,2] y location of message
[M,3] character height in inches
[M,4] angle in degrees to print string (-180 to 180)
[M,5] X,Y coordinate units: 1=plot coord, 2=inches
[M,6] color
[M,7] line thickness.
Example: _pmsgctl = { 10 20 .12 0 1 10 0,10 30 .12 0 1 10 0 }

_pmsgstr Specify additional message text in location corresponding to **_pmsgctl**. Strings are separated by a null byte ("\000")
Example: _pmsgstr = "First message\000Second message"

Windows, Page, and Plot Control

_pagesiz Control size of graph for printer output
_pageshf Shift the graph for printer output
_pbox Draw a box (frame) around window using color
_plotsiz Control plot area size
_plotshf Control plot area position
_protate Rotate the graph 90 degrees
axmargin Control of axes margins (axes lengths)
margin Control graph margins

begwind Window initialization procedure. Initializes global window variables.
Format: begwind;
Input: None.
This procedure must be called before any other window functions are called.

endwind End window manipulation, display graphs. No input.

window Create tiled windows of equal size. Partitions the screen into tiled windows of equal size.
Format: window(row,col,typ);
Input: row Number of rows of windows.
col Number of columns of windows.
typ Window attribute type. If this value is 1, the windows will have a transparent attribute, if 0, the windows will have a non-transparent (blanked) attribute.
The windows will be numbered from 1 to (row x col) beginning from the left topmost window and moving. right. The current window is set to 1 immediately after calling this function. See the **makewind** function for a description of transparentand non-transparent windows.

makewind Creates window with specified size and position and add to list of current windows. **This is important for setting transparency or lack of it in windows overlays.**
Format: makewind(xsize,ysize,xshft,yshft,typ);
Input: xsize Horizontal size of the window in inches.
ysize Vertical size of the window in inches.
xshft Horizontal shift from left edge of screen in inches.
yshft Vertical shift from bottom edge of screen in inches.
typ Window attribute type. If this value is 1, the windows will have a transparent attribute, if 0 the windows will have a normal (non-transparent) attribute.
If the newly created window overlaps any windows previously created, those windows will be clipped to accommodate the new one. This causes the new window to be the topmost window on the screen. This also sets the newly created window to be the current one. A window is normally blanked. That is, the area on the page where it will reside is

blanked and is also clipped to accommodate any windows overlapping it. A transparent window is one which does no clipping to accommodate windows which overlap it, and other windows will not be clipped to accommodate it.

setwind Set to specified window number. Sets the current window to previously created window number.
Format: setwind(n);
Input: n Window number of a previously created window.

nextwind Sets the current window to the next available window number

getwind Get current window number

savewin Save window configuration to a file

loadwin Load a window configuration from a file

Output options

_pnotify Control interactive "draft" mode

_pscreen Control graphics output to screen

_psilent Control final beep

_pzoom Specify zoom parameters

_ptek Name of graphics .TKF file. **This control is extremely important if you want to convert your graphics or to make several in the same program and save the files for later use.**
If _ptek is set to a null string, no file will be created. Otherwise set this to the desired name for the graphics file. You should include the tag .TKF within the filename.
Examples:
_ptek = "test.tkf"; * name the graphics file test.tkf *
_ptek = ""; * do not create a graphics file *

_pqgedit Interactive mouse-driven graphics editor (optional)

graphprt Generate print or conversion file

Line Types

1 Dashed	4 Fine Dots
2 Dots	5 Dots and Dashes
3 Short Dashes	6 Solid

Symbol Types

1 - Circle	8 - Solid Circle
2 - Square	9 - Solid Square
3 - Triangle	10 - Solid Triangle
4 - Plus	11 - Solid Plus
5 - Diamond	12 - Solid Diamond
6 - Inverted Triangle	13 - Solid Inverted Triangle
7 - Star	14 - Solid Star

Color Values

0 - Black	8 - Dark Grey
1 - Blue	9 - Light Blue
2 - Green	10 - Light Green
3 - Cyan	11 - Light Cyan
4 - Red	12 - Light Red
5 - Magenta	13 - Light Magenta
6 - Brown	14 - Yellow
7 - Grey	15 - White

I illustrate three types of graphs: **xy, surface, contour,** and **bar** in **basic11.gss, basic12.gss** and **basic13.gss**.

Example XY(...): **basic11.gss.**

Example: SURFACE(...) and CONTOUR(...); TILED WINDOWS: **basic12.gss.**

Example: BAR(...) OVERLAID WITH 2-D PLOT: **basic13.gss.**
This example illustrates overlay only.

10. MATHEMATICAL FUNCTIONS IN GAUSS

pi Returns the value of pi.
3.1415927 ....
Consequently, pi or Pi is a reserved word; you cannot use it except to mean this number.

All trigonometric functions take or return values in radian units (fractions of 2π). All mathematical functions are calculated in double precision, with the exception of the Bessel functions and the Gamma function. These are calculated to roughly single precision.

Some examples and graphs are presented in basic14 below.

a. Trigonometric Functions

arccos(x) Inverse cosine.
Input: x NxK matrix.
Output: y NxK matrix containing the angle in radians whose cosine is x.
If x is complex, arccos is defined for all values. If x is real, arccos is defined only for abs(x) <= 1. If any elements of x are out of this range, the procedure will terminate with an error message.

arcsin Inverse sine. Similar to arccos.

atan(x) Inverse tangent.
Input: x NxK matrix.
Output: y NxK matrix, the arctangent of x.
For real x, the elements of y are radian values in the interval:
-pi/2 <= y <= pi/2
For complex x, the arctangent is defined everywhere except i and -i. If x is complex, y will be complex.

atan2(y,x) Angle whose tangent is y/x. Computes the angle between the positive x axis and the ray which originates at (0,0) and passes through (x,y).
Input: y NxK matrix.
x LxM matrix, element by element conformable with y.
Output: z max(N,L) by max(K,M) matrix.
The elements of z are radian values in the interval:
-pi <= y <= pi

cos Cosine.

cosh(x) Hyperbolic cosine. $\cosh(x) = \frac{e^x + e^{-x}}{2}$.

sin Sine.

sinh(x) Hyperbolic sine. $\sinh(x) = \frac{e^x - e^{-x}}{2}$.

tan Tangent.

tanh(x) Hyperbolic tangent. tanh(x) = sinh(x)/cosh(x).

b. Other Mathematical Functions

Bessel functions are widely used in the numerical solution of differential equations and are themselves solutions to the differential equation:

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - n^2) w = 0.$$

Solutions are Bessel functions of the first kind, $J_{\pm n}(z)$, of the second kind, $Y_n(z)$, and of the third kind, the so-called Hankel functions. When ν is an integer, these functions have simpler properties than when is unrestricted.

besselj Bessel function, 1st kind, $J_{\pm n}(z)$.

Format: y = besselj(ν,x);

Input: ν NxK matrix, the order of the Bessel function. Nonintegers are truncated to an integer.

x LxM matrix, element by element conformable with ν.

Output: y max(N,L) by max(K,M) matrix.

bessely Bessel function, 2nd kind, $Y_n(z)$.

Format: y = bessely(ν,x);

Input: ν NxK matrix, the order of the Bessel function. Nonintegers are truncated to an integer.

x LxM matrix, element by element conformable with ν.

Output: y max(N,L) by max(K,M) matrix.

mbesselei
mbesselei0
mbesselei1
mbesseli
mbesseli0
mbesseli1

Advanced functions related to Bessel functions.

exp(x) Exponential function. x is restricted to be real.

gamma(x) Gamma function:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$ = x!, when x is a positive integer.

Input: x NxK matrix

Output: y NxK matrix

For each element of x, **gamma** returns the above integral.All elements of x must be positive and less than or equal to 169. Values of x greater than 169 cause an overflow. The natural log of gamma is often what is required and it can be computed without the overflow problems of gamma. **lnfact** can be used to compute log gamma.

gammaii(a,p) Compute the inverse incomplete Gamma function. This function is proportional to the chi-square distribution.
Use **cdfgam** instead; see below **STATISTICAL DISTRIBUTIONS.

ln Natural log.

lnfact(x) Natural log of factorial function. The first derivative of this function, $\Gamma'(x)/\Gamma(x)$, is called the digamma function and occurs in many statistical applications.
Input: x NxK matrix. All elements must be non-negative.
Output: y NxK matrix containing the natural log of the factorials of each of the elements of x.
For integer x in the range from 1 to 172, this is (approximately) ln(x!). However, the computation is done using Stirling's formula for x > 1, and the function is defined fornon-integer x. For $0<x\leq 1$, a power series expansion of $\ln\Gamma(x)$ is used. In most formulae in which the factorial operator appears, it is possible to avoid computing the factorial directly, and to use **lnfact** instead. The advantage of this is that **lnfact** does not have the overflow problems that ! has. For x >= 1, this function has at least 6 digit accuracy, for x > 4 it has at least 9 digit accuracy, and for x > 10 it has at least 12 digit accuracy. For 0 < x < 1, accuracy is not known completely but is probably at least 6 digits.Sometimes log gamma is required instead of log factorial.These are related by:

ln(gamma(x)) = lnfact(x - 1);

log Log base 10.

sqrt(x) Square root. If x is a negative real number, sqrt will return a complex number.

Some of these functions are graphed by **basic14.gss.**

11. CHARACTERISTIC ROOTS AND VECTORS; MATRIX DECOMPOSITIONS; SOLUTION OF LINEAR EQUATIONS.

balance Balances a square matrix.[2]
Format: { b,z } = balance(x);
Input: x KxK matrix.
Output: b KxK matrix, balanced matrix.
z KxK matrix, diagonal scale matrix.
balance returns a balanced matrix b and another matrix z with scale factors in powers of two on its diagonal. b is balanced in the sense that the absolute sums of the magnitudes of elements in corresponding rows and columns are nearly equal. **balance** can be used to scale matrices to improve the numerical stability of the calculation of their characteristic values. It may also be useful in the solution of matrix equations, and it can be used to scale the data to improve the performance of the applications module CML (constrained maximum likelihood).

chol Computes the Cholesky decomposition of a symmetric, positive definite matrix.
Format: T = chol(x);
Input: x NxN matrix (x should be positive definite for this to work properly, but **chol** does not check).
Output: T NxN matrix upper triangular matrix such that x = T'T, the Cholesky decomposition of x.

[2] The sensitivity of the results of calculations of characteristic roots (values) and vectors is generally proportional to the Euclidean norm of the matrix, i.e., the square root of the sum of squares of the elements. The idea of balancing is to make the corresponding rows and columns of of the matrix involved have comparable norms, thus reducing the overall norm of the matrix while leaving the characteristic values and vectors unchanged. A symmetric matrix is already balanced. See Press, *et al., op. cit.*, pp. 365 - 367.

This command is essential in obtaining random vectors distributed according to a multivariate normal distribution with specified mean and variance-covariance matrix from a number of independent standard normal random deviates.
Nota bene, **chol** does NOT check to see that the matrix is symmetric. **chol** looks only at the upper half of the matrix including the principal diagonal. If the matrix x is symmetric but not positive definite, either an error message or an error code is generated and your progam will be terminated

choldn Performs a Cholesky "downdate" of one or more rows on an upper triangular matrix. (Subtracting one or more rows.)

Format: T = choldn(C,x);
Input: C KxK upper triangular matrix. C should already be a Cholesky factorization.
x NxK matrix, the rows to "downdate" C with.
Output: T KxK upper triangular matrix, the "downdated" matrix.

choldn(C,x) is equivalent to **chol(C'C - x'x)**, but **choldn** is numerically much more stable. It is possible to render a Cholesky factorization non-positive definite with **choldn**. You should keep an eye on the ratio of the largestdiagonal element of T to the smallest--if it gets very large, T may no longer be positive definite. This ratio is a rough estimate of the condition number of the matrix.

cholup Performs Cholesky "update" on an upper triangular matrix. Format same as **choldn. cholup(C,x)** is equivalent to **chol(C'C + x'x)**, but **cholup** is numerically much more stable.

cholsol Solves a system of equations given the Cholesky factorization of a matrix.

Format: x = cholsol(b,C);
Input: b NxK matrix.
C NxN matrix.
Output: x NxK matrix.

C is the Cholesky factorization of the matrix of a linear system of equations, A. x is the solution for Ax = b. b can have more than one column. **cholsol(eye(N),C)** is equivalent to **invpd(A)**. Thus, if you have the Cholesky factorization of A anyway, **cholsol** is the most efficient way to obtain the inverse of A.

cond Computes condition number of a matrix. This is a very important command. See **svd**, **svd1**, **svd2**, **svdcusv**, **svds**, and **svdusv**, where the singular value decomposition is further discussed below.[3]

Format: c = cond(x);
Input: x NxP matrix.
Output: c scalar, an estimate of the condition number of x. This equals the ratio of the largest singular value to the smallest. If the smallest singular value is zero or not all of the singular values can be computed the return value is 1.0e+300.

crout Computes the Crout decomposition of a square matrix, X = L*U, without row pivoting.[4]

---

[3] The condition number of a matrix is derived from the so-called *singular value decomposition* of a matrix: Any m × n, m≥n, A can be decomposed into the product of an m × n column-orthogonal matrix U, an n × n diagonal matrix W with positive or zero elements, and the transpose of an n × n orthogonal matrix V. The diagonal elements of W are called the *singular values* of A. The singular value decomposition of a matrix is discussed, *inter alia*, in J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis, 2nd. ed.*, New York, Springer-Verlag, 1993. Press, *et al., op. cit.*, pp. 52 - 64, give a very comprehensive discussion of their importance in computational mathematics. Formally, the *condition number* of a square matrix A (if A is not square some condition numbers must be zero) is defined as the ratio of the largest diagonal element of W to the smallest. A square matrix is singular if its condition number is infinite, and it is *ill-conditioned* if its condition number is too large (e.g., approaching the limit of the computer's floating point precision). The GAUSS procedures **svd, svd1, svd2, svdcusv, svds**, and **svdusv**, all compute singular value decompositions or matrices associated with such decompositions. The GAUSS procedure **cond** simply computes the condition number; if the matrix is not square, 10^{300} is returned. **cond** can be used to check whether your results from solving a system of equations might be inaccurate.

Format: y = crout(x);
Input: x NxN real nonsingular matrix.
Output: y NxN matrix containing the lower (L) and upper (U) matrices of the Crout decomposition of x. The main diagonal of y is the main diagonal of the lower matrix L. The upper matrix has an implicit main diagonal of ones. Use the procedures **lowmat** and **upmat1** to extract the L and U matrices from y. (See section 4.c(5) above.) Since **crout** does not use row pivoting, **croutp** is mostly used in practice, but it's harder to see what it's doing.

croutp Computes Crout decomposition with row pivoting. Same format as **crout** except that the output is y (N+1)xN matrix containing the lower (L) and upper (U) matrices of the Crout decomposition of a permuted x. **The (N+1)th row of the matrix y gives the row order of the y matrix**. The matrix must be reordered priorto extracting the L and U matrices. Use the commands **lowmat** and **upmat1** to extract the L and U matrices from the reordered y matrix.

det Returns the determinant of a square matrix.
Format: y = det(x);
Input: x NxN square matrix.
Output: y scalar, determinant of x.
x may be any valid expression that returns a square matrix. **detl** works on the last matrix returned by a matrix decomposition program and can be much faster.

detl Returns the determinant of the last matrix that was passed to one of the intrinsic matrix decomposition routines.
Format: y = detl;
Whenever one of the following functions is executed, the determinant of the matrix is also computed and stored in a system variable. **detl** returns the value of that determinant. Because the value has been computed in a previous instruction, this requires no additional computation.

chol(x);
crout(x);
croutp(x);
det(x);
inv(x);
invpd(x);
solpd(x);
y/x

GAUSS has a great many built-in functions for the computations of eigen- or characteristic values[5]; here are two (GAUSS's terminology is *eigenvalues* and *eigenvectors;* I will use it because it makes the names of the commands easier to remember):

eig Computes the eigenvalues of a general matrix.
Format: va = eig(x);
Input: x NxN matrix.
Output: va Nx1 vector.

eigv Computes the eigenvalues **and eigenvectors** of a general matrix.
Format: { va,ve } = eigv(x);
Input: x NxN matrix.

---

[4] See Stoer and Bulirsch, *op. cit.*,pp. 174 - 176, for a discussion of the Crout form of the Gaussian elimination algorithm and why row pivoting is important.

[5] Recall that the characteristic values of a square matrix A, n by n, are the roots of the *characteristic polynomial* $p(\lambda) = \det(\lambda I_n - A)$. There are n of them, not necessarily distinct; none are equal to zero if A is nonsingular; all are positive and real if A is positive definite. These values ensure that the system $Ax = x\lambda$ is solvable. If A is positive definite and symetric, the vectors which solve $Ax = \lambda x$ are independent and orthongonal. So if Λ is the diagonal matrix having the positive λ's along the diagonal, the matrix X composed of the solutions to $Ax = \lambda x$ "diagonalizes" A: $X'AX = \Lambda$, or $A = X\Lambda X'$. Indeed, A need not be positive definite or even nonsingular for this result to hold.

Output: va Nx1 vector, the eigenvalues.
ve NxN matrix, the eigenvectors.

If the eigenvalues cannot all be determined, va[1] is set to an error code. The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first. The columns of ve contain the eigenvectors of x in the same order as the eigenvalues. The eigenvectors are not normalized.

See also **eigh, eighv, eigcg, eigcg2, eigch, eigrg, eigrg2, eigrs,** and **eigrs2. eigrs** and **eigrs2** are the most usefurl for us since they are the same as **eig** and **eigv** but restricted to real symmetric matrices. Check the manual or on-line help.

hess Computes the upper Hessenberg form of a matrix. The Hessenberg form is an intermediate step in computing characteristic values. It also is useful for solving certain matrix equations that occur in control theory. **The command hess should be carefully distinguished from the command hessp, which computes the hessian of a single valued function and which we will use extensively in applications of maximum-likelihood methods.**

inv Returns the inverse of an invertible matrix.
Format: y = inv(x)
Input: x NxN matrix.
Output: y NxN matrix containing the inverse of x.
x can be any expression that returns a square matrix.
If the input matrix is not invertible, inv either terminates the program with an error message or returns an error code which can be tested for with the **scalerr** function.[6] This depends on the setting of **trap**: **trap 1;** = return error code 50 and **;trap 0** = terminate with error message "Matrix singular" **and stop program**. If you want to prevent the program from stopping set **trap 1;**.
Positive definite matrices can be inverted by inv. However, for symmetric, positive definite matrices (such as moment matrices), **invpd** is about twice as fast as **inv**.

invpd Returns the inverse of a symmetric, positive definite matrix.
Same format, input and output as **inv**.
If the input matrix is not invertible, **invpd** either terminates the program with an error message or returns an error code which can be tested for with the **scalerr** function. This depends on the setting of **trap**: **trap 1** = return error code 20 and **trap 0 =** terminate with error message "Matrix not positive definite". If the input to **invpd** is not symmetric, it is possible that the function will appear to operate successfully but will not in fact return the right answer.

invswp Computes a generalized Moore-Penrose (sweep) inverse of a not necessarily square matrix.[7]

---

[6] **scalerr** Tests for a scalar error code.
Format: y = scalerr(c);
Input: c NxK matrix, the result of an expression, function or procedure.
Output: y scalar, the value of the error code as an integer.
If the argument is not a scalar error code, 0 is returned.
Error codes in GAUSS are NaN's (Not A Number). These are not just scalar integer values. They are special floating point encodings that the numeric processor recognizes as not representing a valid number. See **error**. **scalerr** can be used to test for either those error codes which are predefined in GAUSS or an error code which the user has defined using **error.** Use the **trap** command to trap errors in functions such as **chol**, **invpd**, /, and **inv**. **trap 1;** = turn trapping on and **trap 0;** = turn trapping off.

[7] Y is the generalized inverse of X if X and Y satisfy the four Moore-Penrose conditions:
1. X*Y*X = X;
2. Y*X*Y = Y;
3. X*Y is symmetric;
4. Y*X is symmetric.

Format: y = invswp(x);
Input: x NxN matrix.
Output: y NxN, the generalized inverse of x.
Even singular matrices (nonsquare matrices and square matrices of less than full rank) have Moore-Penrose inverses. See also **pinv.**

lu Computes LU decomposition, X = L*U of a square matrix with partial (row) pivoting.[8]
Format: { L,U } = lu(x);
Input: x NxN nonsingular matrix.
Output: L NxN "scrambled" lower triangular matrix. This is a lower triangular matrix that has been reordered based on the row pivoting.
U NxN upper triangular matrix.

null Computes orthonormal basis for right null space.
null1 Computes orthonormal basis for right null space.
orth Computes orthonormal basis for column space.
} commands related to **svd.**[9]

pinv Compute the Moore-Penrose pseudo-inverse of a matrix, using the singular value decomposition **svd**.
Format: y = pinv(x);
Input: x NxM matrix.
Returns can be controlled by use of global settings:
_svdtol global scalar, any singular values less than **_svdtol** are treated as zero in determining the rank of the input matrix. The default value for **_svdtol** is 1.0e-13.
_svderr global scalar, if not all of the singular values can be computed **_svderr** will be nonzero. The singular values in **s[_svderr+1], ... s[M]** will be correct.
Output: y MxN matrix that satisfies the four Moore-Penrose conditions.[10]

qqr qr decomposition: returns Q1 and R.
qqre qr decomp: returns Q1, R and a permutation vector E.
qqrep qr decomp. with pivot control: returns Q1, R and E.
qr qr decomposition: returns R.
qre qr decomp: returns R and a permutation vector E.
qrep qr decomp. with pivot control: returns R and E.
qtyr qr decomp: returns Q'Y and R.
qtyre qr decomp: returns Q'Y, R and a permutation vector E.
qtyrep qr decomp. with pivot control: returns Q'Y, R and E.
qyr qr decomp: returns Q * Y and R.
qyre qr decomp: returns Q * Y, R and a permutation vector E.
qyrep qr decomp. with pivot control: returns Q * Y, R and E.
} related to **QR decomposition**.

---

[8] For a careful exposition of the LU decomposition and partial pivoting (interchange of rows) see Press, *et al., op. cit.,* pp. 33 - 38. Partial pivoting means that we don't actually decompose X into L*U but rather a row-wise permutation of X. Of course, we should keep track of what the permutation is and GAUSS does not.
[9] See footnote 3.
[10] See footnote 7.

The QR decomposition of a matrix X:
Given X, there is an orthogonal matrix Q such that Q' * X is zero below its diagonal, i.e.,

(1) $$Q'*X = \begin{bmatrix} R \\ 0 \end{bmatrix},$$ where R is upper triangular. If we partition

(2) Q = [Q1 Q2] , where Q1 has P columns, then

(3) X = Q1 * R, is the **QR decomposition of X**.

If X has linearly independent columns, **R is also the Cholesky factorization of the moment matrix of X, i.e., of X'* X.** If you want only the R matrix, see QR. Not computing Q1 can produce significant improvements in computing time and memory usage. An unpivoted R matrix can also be generated using **cholup**: R = **cholup(zeros(cols(x),cols(x)),x);.** For linear equation or least squares problems, which require Q2 for computing residuals and residual sums of squares, see **olsqr**, and **qtyr.** For most problems an explicit copy of Q1 or Q2 is not required. Instead one of the following, Q'* Y, Q * Y, Q1'* Y, Q1 * Y, Q2'* Y, or Q2 * Y, for some Y, is required. These cases are all handled by **qtyr** and **qyr,** because Q and Q1 are typically very large matrices while their products with y are more manageable.

If N < P the factorization assumes the form:

(4) Q'* X = [R1 R2], where R1 is a PxP upper triangular matrix and R2 is Px(N-P).

Thus Q is a PxP matrix and R is a PxN matrix containing R1 and R2. This type of factorization is useful for the solution of underdetermined systems. However, (unless the linearly independent columns happen to be the initial rows) such an analysis also requires pivoting (see **qre** and **qrep**). **qr** is the same as **qqr** but doesn't return the Q1 matrix. If Q1 is not wanted, qr will save a significant amount of time and memory usage, especially for large problems. If you need the Q1 matrix see the GAUSS function **qqr**. If you need the entire Q matrix call qyr with Y set to a conformable identitymatrix. For linear equation or least squares problems, which require Q2 for computing residuals and residual sums of squares, see **olsqr**.

---

qrsol x = **qrsol**(b,R); Computes the solution of R x = b, where R is upper triangular.

qrtsol x = **qrtsol**(b,L); Computes the solution of R x = b, where L is lower triangular. Generally R will be the R matrix from a QR factorization. It may be used, however, in any situation where R is upper triangular. If b has more than one column, each column will be solved for separately. If you have a lower triangular matrix, apply **qrtsol** to R'.

rank Computes the rank of a matrix, using the *singular value decomposition*, see **svd** below.

Format: k = rank(x);

Input: x NxP matrix.

_svdtol global scalar, the tolerance used in determining if any of the singular values are effectively 0. The default value is 1e-13. This can be changed before calling the procedure.

_svderr global scalar, if not all of the singular values can be computed _svderr will be nonzero. The singular values in s[_svderr+1], ... s[M] will be correct.

Output: k an estimate of the rank of x. This equals the number of singular values of x that exceed a prespecified tolerance in absolute value.

rref Computes reduced row echelon form of a matrix.[11]

Format: y = rref(x);

Input: x MxN matrix.

Output: y MxN matrix containing reduced row echelon form of x.

The tolerance used for zeroing elements is computed inside the procedure using:

tol = maxc(m|n) * eps * maxc(abs(sumc(x')));

where **eps = 2.24e-16;**.

This procedure can be used to find the rank of a matrix. It is not as stable numerically as the singular value decomposition (which is used in the rank function), but it is faster for large

[11] Usually discussed in textbooks on matrix algebra under the heading of "elementary row operations," i.e., operations which do not change the rank of a matrix.

matrices.There is some speed advantage in having the number of rows be greater than tthe number of columns, so you may want to transpose if all you care about is the rank.
The following code can be used to compute the rank of a matrix, using rref:

r = sumc(meanc(abs(y')) .> _rreftol);

where y is the output from **rref**, and **_rreftol** is the tolerance used. This finds the number of rows with any non-zero elements, which gives therank of the matrix, disregarding numeric problems. This code could be placed at the end of rref to convert it into a procedure to compute rank.

schur Computes Schur decomposition of a matrix (real only).[12]

Format: { s,z } = schur(x);
Input: x KxK matrix.
Output: s KxK matrix, Schur form.
z KxK matrix, transformation matrix.
z is an orthogonal matrix that transforms x into s and vice versa. Thus,
s = z * x * z' and x = z' * s * z .

schur computes the real Schur form of a square matrix. The real Schur form is an upper quasi-triangular matrix, that is, it is block triangular where the blocks are 2x2 submatrices which correspond to complex eigenvalues of x. If x has no complex eigenvalues, s will be strictly upper triangular. Use the GAUSS command **schtoc** to convert s to the complex Schur form. (x is first reduced to upper Hessenberg form using orthogonal similiarity transformations, then reduced to Schur form through a sequence of QR decompositions.

schtoc To reduce any 2x2 blocks on the diagonal of the real Schur matrix returned from **schur**. The transformation matrix is also updated.

Format: { schc, transc } = schtoc(sch,trans);
Input: sch real NxN matrix in Real Schur form. I.e. upper triangular except for possibly 2x2 blocks on the diagonal.
trans real NxN matrix, the associated transformation matrix.
Output: schc NxN matrix, possibly complex, strictly upper triangular. The diagonal entries are the eigenvalues.
transc NxN matrix, possible complex, the associated transformation matrix.
Other than checking that the inputs are strictly real matrices, no other checks are made. If the input matrix sch is already upper triangular it is not changed. Small off-diagional elements areconsidered to be zero.
See the source code for the test used, schtoc.src.

solpd Solves a set of positive definite linear equations.

Format: x = solpd(b,A);
Input: b NxK matrix.
A NxN symmetric positive definite matrix.
Output: x NxK matrix, the solutions for the system of equations Ax = b.

solpd uses the Cholesky decomposition to solve the system directly. It is therefore more efficient than using **inv(A)*b. solpd** does NOT check to see that the matrix A is symmetric. **solpd** looks only at the upper half of the matrix including the principal diagonal. If the matrix A is not positive definite, the current trap state determines what the program returns: **trap 1** return scalar error code 30 or **trap 0** terminate with an error message.One obvious use for this function is to solve for least squares coefficients. The effect of **solpd** is thus similar to that of the / operator.

svd Computes the singular values of a matrix.[13]

---

[12] The Schur form is defined by the Schur Triangularization Theorm: For every square (real) matrix A, there exists an orthogonal transformation C such that C'AC = T, where T is upper triangular, with the characteristic vlues of A along the main diagonal in any prescribed order. T is the Schur form.

[13] See footnote 3. If the matrix A is square, say n by n, The matrices U, V, and W of the SVD are all square of the same size. Their inverses are all easy to compute: U and V are orthogonal matrices, so their inverses are equal to their transposes. W is diagonal and its inverse is the matrix with the w_j's pf the SVD along the

svd1 Computes singular value decomposition, X = USV'.
svd2 Computes svd1 with compact U.
See also **svdcusv, svds**, and **svdusv**

The above commands are illusstrated, in part, in **basic15.gss.**

12. POLYNOMIALS.

polychar Computes characteristic polynomial of a square matrix, i.e. the coefficients in $|X - zI| = 0$.

Format: c = polychar(x);
Input: x NxN matrix.
Output: c N+1x1 vector of coefficients of the Nth order characteristic polynomial of x:

p(z)=c[1,1]*z^n + c[2,1]*z^(n-1) + ... +c[n,1]*z + c[n+1,1];

The coefficient of z^n is set to unity (c[1,1]=1).

polyeval To evaluate polynomials. Can either be one or more scalar polynomials, or a single matrix polynomial.

Format: y = polyeval(x,c);
Input: x 1xK or NxN; that is, x can either represent K separate scalar values at which to evaluate the (scalar) polynomial(s), or it can represent a single NxN matrix.
c P+1xK or P+1x1 matrix of coefficients of polynomials to evaluate. If x is 1xK, then c must be P+1xK. If x is NxN, c must be P+1x1.That is, if x is a matrix, it can only be evaluated at a single set of coefficients.
Output: y Kx1 vector (if c is P+1xK) or NxN matrix (if c is P+1x1 and x is NxN):

y = (c[1,.].*x^p + c[2,.].*x^(p-1) + ... + c[p+1,.])';

In both the scalar and the matrix case, Horner's rule is used to do the evaluation. In the scalar case, the function recsercp is called (this implements an elaboration of Horner's rule).

polyint Calculates a Nth order polynomial interpolation or extrapolation of x on y given the vectors xa and ya and the scalar x. The procedure uses Neville's algorithm to determine an up to Nth order polynomial and an error estimate.[14]

Format: y = polyint(xa,ya,x);

Input: xa Nx1 vector, x values.

ya Nx1 vector, y values.

x scalar, x value to solve for.

_poldeg global scalar, the degree of polynomial required, default 6.
Output: y result of interpolation or extrapolation.

---

diagonal. Hence, if the SVD is A = UWV', A^{-1} = V(diag($1/w_j$))U'. This formulation can go wrong as a computational device if one or mor of the w_j's is so small that its reciprocal is dominated by round-off error. As indicated above, the *condition number* of a square matrix, which is the ratio of the largest of the w_j's to the smallest, gives a measure of the problem. Since the SVD is a fundamental decomposition, the problem exists for any method of inversion. See Press, *et al., op. cit.,* pp. 54 - 64.

[14] See the excellent discussion of interpolation and extrapolation in Press, *et al., op. cit.,* Chapter 3, pp. 77 - 101. Neville's algorithm is discussed pp. 81 - 82.

_polerr global scalar, interpolation error.

Polynomials above degree 6 are not likely to increase the accuracy for most data. Test **_polerr** to determine the required **_poldeg** for your problem.

polymult Multiplies two polynomials together.

Format: c = polymult(c1,c2);

Input: c1 d1+1x1 vector containing the coefficients of the first polynomial.
c2 d2+1x1 vector containing the coefficients of the second polynomial.

Output: c d1+d2x1 vector containing the coefficients of the product of the two polynomials.

If the degree of c1 is d1 (eg, if d1=3, then the polynomial corresponding to c1 is cubic), then there must be d1+1 elements in c1 (e.g. 4 elements for a cubic). Thus, for instance the coefficients for the polynomial 5*x.^3 + 6*x + 3 would be: c1=5|0|6|3.**(Note that zeros must be explicitly given if there are powers of x missing.)**

conv Computes the convolution of two vectors (this is similar to polynomial multiplication).

Format: c = conv(b,x,first,last);

Input: b Nx1 vector.
x Lx1 vector.
first scalar, the first convolution to compute.
last scalar, the last convolution to compute.

Output: c Qx1 result, where Q = (l - first + 1ast). If first is 0, the first to the last convolutions are computed. If last is 0, the first to the last convolutions are computed. If first and lastare both zero, all the convolutions are computed. If x and b are vectors of polynomial coefficients, this is the same as multiplying the two polynomials. See also **fft**, fast fourier transforms, below.

polymake To compute the coefficients of a polynomial, given the roots of the polynomial. (Restricted to real roots).

Format: c = polymake(r);

Input: r Nx1 vector containing roots of the desired polynomial.

Output: c N+1x1 vector containing the coefficients of the Nth order polynomial with roots r:

p(z)=c[1]*z^n + c[2]*z^(n-1) + ... + c[n]*z + c[n+1];

The coefficient of z^n is set to unity (c[1]=1).

polymat Returns a matrix containing the powers of the elements of x from 1 to p.

Format: y = polymat(x,p);

Input: x NxK matrix.
p scalar, positive integer.

Output: x Nx(p*K) matrix containing powers of the elementsof x from 1 to p. The first K columns will contain first powers, the second K columns contain the second powers, and so on.

polyroot Computes roots of polynomial from coefficients c.

Format: r = polyroot(c);

Input: c N+1x1 vector of coefficients of an Nth order polynomial:
p(z)=c[1]*z^n + c[2]*z^(n-1) + ... + c[n]*z + c[n+1];
Zero leading terms will be stripped from c. When thatoccurs the order of the r will be the order of the polynomial after the leading zeros have been stripped. c[1] need not be normalized to unity.

Output: r Nx2 vector containing the roots of c.

basic16.gss illustrates the application of these polynomial routines:

13. FOURIER TRANSFORMS

Fourier transforms are important, indeed essential, in time series analysis. GAUSS supports a great variety of Fourier transform routines. Most of these are *fast fourier transforms*, which require the number of data points to be an exact power of 2 or of 3, 5, and 7.These are: **fft**, **rfft**, and **rfftp** which require the dimensions of the input matrix to be powers of 2. **fftn**, **ffti**, **rfftn**, **rfftnp**, **rffti**, and **rfftip** which allow them to be products of 2, 3, 5, and 7.**rfftnp** and **rfftp** return only the positive frequencies and Nyquist frequencies, as the negative frequencies are often not needed in most applications.[15] The two GAUSS routines which are of primary importance for us are:

dfft Computes a complex discrete Fourier transform.

Format:	y = dfft(x);		x and y are complex
	{ yr,yi } = dfft(xr,xi);		xr , xi, yr and yi are real
Input:	x	Nx1 complex vector or	
	xr	Nx1 vector, real part	
	xi	Nx1 vector, imaginary part	
Output:	y	Nx1 complex vector or	
	yr	Nx1 vector, real part.	
	yi	Nx1 vector, imaginary part.	

The transform is divided by N.

dffti Computes an inverse complex discrete Fourier transform.
Same format, input and output as above.

basic17.gss illustrates.

14. RANDOM NUMBER GENERATION AND STATISTICAL DISTRIBUTIONS

(a) Random Number Generators

For a geneneral discussion of these matters, you should consult M. Nerlove, "Notes on Monte Carlo, Bootstrapping and Estimation by Simulation," section 1, "Random Number Generation," available through the link on my homepage, to **Short course on Monte Carlo Methods, Resampling Techniques, and Simulation Methods for Estimation in Econometrics.** GAUSS has the following on-line commands for random number generation:

Random Number Generators: Uniform and Normal

rndn Creates a matrix of normally distributed random numbers.

rndu Creates a matrix of uniformly distributed random numbers.

Format:	y = rndn(r,c);	
Input:	r	scalar, row dimension.
	c	scalar, column dimension.
Output of **rndn**:	y	r×c matrix of normal random numbers having a mean of 0 and a standard deviation of 1.
Output of **rndu**:	y	r×c matrix of uniform random numbers between 0 and 1. r and c are truncated to integers if necessary.

rndn and **rndu** are based upon the same linear congruential uniform random number generator. [16] This generator is initially seeded using the system clock when GAUSS starts

---

[15] You can read what I have to say about Fourier Transforms and analysis, Appendix C, pp. 380 - 412, of M. Nerlove, D. M. Grether and J. L. Carvalho, *Analysis of Economic Time Series*, Revised Edition, San Diego: Academic Press, 1995. Press, *et al., op. cit.,* Chapter 12, pp. 381 - 453, give a very complete discussion, primarily but not exclusively, from a computational point of view.

[16] See Nerlove, *op. cit.,* 1997. A recursive relation called the *linear congruential generator*:

$$u_{i+1} = (au_i + c)(\mathrm{mod}\, m),$$

up. It can be reseeded using **rndseed**, **rndns** and **rndus**. The other parameters of the generator can be changed using **rndcon**, **rndmod** and **rndmult**. The seed is automatically updated each time a random number is generated (see **rndcon**). Thus, if GAUSS is allowed to run for a long time, and if large numbers of random numbers are generated, there is a possibility of recycling.This is a 32-bit generator, though, so the range is sufficient for most

Random Number Generator Controls for use with **rndu** and **rndn.**

rndcon Resets the constant parameter of the uniform and normal random number generators.
Format: rndcon c;
rndcon resets the constant parameter, c, of the linear congruential uniform random number generator . All the parameters of this generator must be integers in the range: 0 < prm < 2^31-1. The procedure used to generate the uniform random numbers is as follows. First the current seed is used to generate a new seed:

new_seed = (a*seed + c) % m (where % is the mod operator)

Then a number between 0 and 1 is created by dividing the new seed by m:

x = new_seed/m

The default is c = 0.

rndmod resets m. The default is m = 2^31-1.

rndmult resets a. The default is a = 397204094.

rndseed resets seed. The initial seed is generated by the system clock when GAUSS is invoked. Thereafter, the seed is updated each time **rndn** or **rndu** is called.

rndns Creates a matrix of normally distributed random numbers using a specified seed.

rndus Creates a matrix of uniformly distributed random numbers using a specified seed.

If you are running a very long Monte Carlo, you may want to break the run into pieces. In this case, save the last random number generated and use it multiplied by m as the new seed for **rndus** or **rndns.** If you are teaching and want to give an exercise that's easy to grade, give everyone the same seed so that each will generate the same sequence of "random numbers."

Other generators in GAUSS.

rndgam Computes Gamma pseudo-random numbers.[17]
Format: x = rndgam(r,c,alpha);

Input:	r	scalar, number of rows of resulting matrix.
	c	scalar, number of columns of resulting matrix.
	alpha	r×c matrix, or rx1 vector, or 1Xc vector, or scalar, shape argument for gamma distribution.
Output:	x	r×c matrix, gamma distributed random numbers.

---

where m is called the *modulus*, and a and c are positive integers called the *multiplier* and the *increment* respectively. The reccurrence (2) will eventually repeat itself, but if m, a and c are properly chosen the period of the generator will be m, the maximum possible. The number m is usually chosen to be about the length of a machine word, making the generator machine dependent. The GAUSS manual remarks that the generator assumes a 32-bit machine (p.1525), which means m should be about 2^{32}. This is consistent with the restriction in **rndus** that the user supplied seed must be in the range 0<seed<2^{31}-1 (p.1523). The example used there chooses 3937841 as the seed. The default value for m is in fact 2^{31}-1 = 2147483647 (p. 1519).The default values are c=0, a=397204094 and m=2147483647. But you can set your own parameters and starting seed with **rndcon**, **rndmod**, and **rndseed** (p.1519).

[17] The gamma distribution for integer parameter $\alpha > 0$ is the waiting time to the αth event for a Poisson process having mean 1. If $\alpha = 1$, the gamma distribution degenerates to the exponential distribution. For parameter α, a gamma deviate has probbability $p(x;a)dx$ of occurring with a value between x and x+dx,

where $$p(x;a) = \frac{x^{a-1}e^{-x}}{\Gamma(a)}dx, \quad x > 0.$$

The properties of the pseudo-random numbers in x are:

E(x) = alpha, Var(x) = alpha, x > 0, alpha > 0.

To generate gamma(alpha,theta) pseudo-random numbers where theta is a scale parameter, multiply the result of rndgam by theta.Thus

z = theta * rndgam(1,1,alpha);

has the properties

E(z) = alpha * theta, Var(z) = alpha * theta ^ 2, z > 0, alpha > 0, theta > 0.

rndp Computes Poisson pseudo-random numbers.[18]

Format: x = rndp(r,c,lambda);

Input:
- r scalar, number of rows of resulting matrix.
- c scalar, number of columns of resulting matrix.
- lambda rxc matrix, or rx1 vector, or 1Xc vector, or scalar, shape argument for Poisson distribution.

Output:
- x rxc matrix, Poisson distributed random numbers.

The properties of the poisson pseudo-random numbers in x are

E(x) = lambda, Var(x) = lambda x = 0, 1, ..., lambda > 0.

rndnb Computes negative binomial pseudo-random numbers.[19]

Format: x = rndnb(r,c,k,p);

Input:
- r scalar, number of rows of resulting matrix.
- c scalar, number of columns of resulting matrix.
- k r×c matrix, or rx1 vector, or 1×c vector, or scalar, "event" argument for negative binomial distribution.
- p r×c matrix, or rx1 vector, or 1×c vector, or scalar,"probability" argument for negative binomial distribution.

Output:
- x rxc matrix, negative binomial distributed random numbers.

The properties of the negative binomial pseudo-random numbers in x are:

$$E(x)=\frac{kp}{1-p},\quad Var(x)=\frac{kp}{(1-p)^2},\quad x = 0,1,\ldots,\quad k>0,\quad 0<p<1.$$

rndbeta Computes beta pseudo-random numbers.[20]

---

[18] The Poisson and gamma distributions are related: The Poisson distribution (unit rate) is the probability that λ, an integer, events occur within the interval of time x; on the other hand, the gamma distribution, discussed in the preceeding footnote, is the probability of awaiting time between x and x+dx to the mth event; so integrate gamma between x-ε and x+ε and let $\varepsilon \to 0$ to obtain the probability of x events,

$$\frac{x^{l}e^{-l}}{x!},\quad x=0,1,2,\ldots.$$

[19] A random variable x has a *negative binomial distribution* with density

$$p(x;k,p)=\begin{cases}\binom{k+x-1}{x}p^k q^x=\binom{-k}{x}p^k q^x, & q=1-p,\ \text{for } x=0,1,\ldots\\ 0, & \text{otherwise.}\end{cases}$$

When k = 1 the negative binomial specializes to the geometric distribution:

$$p(x;p)=\begin{cases}p(1-p)^x, & \text{for } x=0,1,\ldots\\ 0, & \text{otherwise.}\end{cases}$$

The importance of the negative binomial derives from the importance of the geometric distribution and the fact that the sum of k iid geometrically distributed RVs has a negative binomial distribution.

[20] A continuous RV x taking on values in the interval (0,1) has a beta distribution if its density is

Format: x = rndbeta(r,c,a,b);
Input: r scalar, number of rows of resulting matrix.
c scalar, number of columns of resulting matrix.
a r×c matrix, or rx1 vector, or 1Xc vector, or scalar, first shape argument for beta distribution.
b r×c matrix, or rx1 vector, or 1Xc vector, or scalar, second shape argument for beta distribution.
Output: x r×c matrix, beta distributed random numbers.

The importance of the beta distribution arises because of the great variety of shapes it can assume.

rndvm Computes von Mises pseudo-random numbers.

The von Mises distribution is sometimes called the circular normal. It has no importance in econometrics as far as I know.

David Baird's generators.[21] (When you use Baird's programs, either download them from the GAUSS archive site or copy them from my illustrative programs below and attach them as procedures to your own program.)

rndt Random numbers from Students T distribution.
Format: x = rndt(r,c,n)
Input: r row size returned matrix
c column size returned matrix
n matrix of degrees of freedom for t distribution elemet by
Output: x an r×c matrix of random numbers ~ t(n)

rndf Random numbers from F distribution.
Format: x = rndf(r,c,n1,n2)
Input: r row size returned matrix
c column size returned matrix
n1 matrix of numerator degrees of freedom for F distribution, element by element conformable with the r×c output matrix
n2 matrix of denominator degrees of freedom for F distribution, element by element conformabler with the output matrix.

b. Density Functions

Generally, it is pretty easy to compute probability densities from the known functional form; what is more of a problem are the cumulative distributions which involve integrals. Nonetheless GAUSS has a few built-in functions:

pdfn Computes the standard normal probability density function.
Format: y = pdfn(x);
Input: x NxK matrix.
Output: y NxK matrix containing the standard normal probability density function of x.

$$f(x;a,b) = \frac{1}{B(a,b)} x^{a-1}(1-x)^{b-1}, \text{ for } x \in (0,1), = 0, \text{ otherwise, where}$$

$$B(a,b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx \text{ is the so-called } \textit{beta function } = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

When a = b = 1, the beta distribution reduces to the uniform over the interval (0,1). The cumulative is called the *incomplete beta function* and computed using the GAUSS procedure **cdfbeta.**

[21] David Baird, AgResearch, PO Box 60, Lincoln, NEW ZEALAND BairdD@AgResearch.CRI.NZ. Programs available at http://netec.wustl.edu/~adnetec/CodEc/GaussAtAmericanU/PDF/PDF.HTM.

pdfn does not compute the joint normal density function. Instead, the scalar normal density function is computed element by element. y could be computed by the following code: **y = 1/sqrt(2*pi))*exp(-(x.*x)/2);**

lnpdfn Computes normal log-probabilities.

Format: z = lnpdfn(x);

Input: x NxK matrix, data.

Output: z NxK matrix, log-probabilities.

lnpdfn does not compute the log of the joint normal pdf.

lnpdfmvn Computes multivariate normal log-probabilities.

Format: z = lnpdfmvn(x,s);

Input: x NxK matrix, data.

s K×K matrix, covariance matrix.

Output: z N×1 vector, log-probabilities.

GAUSS does not appear to have a procedure for computing joint probabilities from a multivariate normal distribution (unlogged); however, it is simple to write a transformation using the Cholsky decompositin of a positive definite matrix to transform n independent standard normal variables into an n-variate normal with arbitrary variance-covariance matrix and mean vector.

David Baird (*loc. cit.*) also has procedures for the following densities:

pdft Students t Probability Density function

Format: d = pdft(x,n)

Input: x matrix of T values

n matrix of degrees of freedom for t distribution

Output: d matrix of density of T(N) function at X

pdff Probability Density function of F distribution

Format: d = pdff(f,n1,n2)

Input: x r × c matrix of F ratios (F > 0)

n1 matrix of numerator degress of freedom (n1 > 0)

n2 matrix of denominator degress of freedom (n2 > 0)

(n1 & n2 must be element by element conformable with x)

Output: d r × c matrix of densities of F(n1,n2)

pdfchi Chi squared Probability Density function with n degrees of freedom

Format: d = pdfchi(x,n)

Input: x matrix of chi-squared values (x > 0)

n matrix of degrees of freedom (n > 0) (conformable with x)

Output: d matrix of density of Chi square(n) at x.

pdfb Binomial Probability Distribution function

Format: y = pdfb(x,p,n)

Input: x r × c matrix of number of successes (0 < x < n)

p matrix of probability of success (0 < p < 1)

n matrix of number of trials (n > 0)

(p & n must be element by element conformable with x)

Output: y r × c matrix of probabilities such that Pr(x = ξ) = y where ξ~ Binomial(p,n).

pdfp Poisson Probability Distribution function

Format: y = pdfp(x,m)

Input: x r × c matrix of number of successes (n >= 0)

m matrix of mean number of successes (m > 0)

(m must be element by element conformable with x)

Output: y r × c matrix of probabilities such that Pr(x = ξ) = y where ξ ~ Poisson(m).

pdfnb Negative Binomial Probability Distribution function

Format: y = pdfnb(x,p,n)
Input: x r x c matrix of number of successes (x >= 0)
p matrix of probability of success (0 < p < 1)
n Negative-Binomial parameter (n > 0)
(p & n must be element by element conformable with x)
Output: y r × c matrix of probabilities st. pr(x = ξ) = y where ξ ~ Negative-Binomial(p,n)

The mean, variance of the Negative Binomial are:

m = n(1-p)/p; v = n(1-p)/(p^2).

The Negative Binomial is also commonly parameterized as nb(p,n), where p = m. As n tends to infinity and p to 1, such that n(1-p)/p = m (m a constant) then the Negative Binomial tends to a Poisson distribution with mean m. Baird's procedure uses the explicit definition for moderate values of n and x and the relationship with the beta distribution for large values of n or x.

pdfgam Standardized Gamma Probability Density function
Format: d = pdfgam(x,a)
Input: x matrix of Gamma values
a matrix of shape parameter for Gamma distribution
Output: d matrix of density of Gamma(a) function at x

c. Cumulative Distribution Functions

GAUSS has the following built-in procedures for evaluating cumulative distribution functions (cdf's) or the complement, 1 - cdf. The computation of the complement is performed directly rather than by using the relation 1 - cdf because there are fewer problems with round-off error and because the complement is most frequently used in statistical applications.[22] I've also included the multivariate and inverse distribution functions which GAUSS supports. Additional references to David Baird's programs (*loc. cit.)* are interspersed and marked with a preceeding asterisk.

Cumulative and complements:

*** cdfb** Binomial Cumulative Distribution function
Format: y = cdfb(x,p,n)
Input: x r × c matrix of number of successes (0 < x < n)
p matrix of probability of success (0 < p < 1)
n matrix of number of trials (n > 0)
(p & n must be element by element conformable with x)
Output: y r × c matrix of probabilities st. Pr(ξ≤x) = y where ξ ~ Binomial(p,n)

This program uses the relationship between the Binomial distributionand the F distribution. (Press, *et al., op. cit.,* p. 169.)The approximation has an accuracy of 6 decimal places but is more stablefor large values of N than using the summation of individual Binomial terms.

cdfbeta Computes the incomplete beta function (i.e. the cumulative distribution function of the beta distribution).
Format: y = cdfbeta(x,a,b);
Input: x N×K matrix
a L×M matrix, element by element conformable with x.

[22] GAUSS also lists
erf Computes Gaussian error function.
erfc Computes complement of Gaussian error function.
These are basically cumulative normals and used in physical applications:

$$erf\left(\frac{x}{\sqrt{2}}\right) = 2cumnormden(x) - 1, \text{ x > 0.}$$

b P×Q matrix, element by element conformable with x and a.

Output: y max(N,L,P) by max(K,M,Q) matrix y is the integral from 0 to x of the beta distribution with parameters a and b. Allowable ranges for the arguments are: 0 <= x <= 1, a > 0 b > 0. -1 is returned for those elements with invalid inputs.

cdfchic Computes complement of cdf of chi-square.

Format: y = cdfchic(x,n);

Input: x N×K matrix.

n L×M matrix, element by element conformable with x.

Output: y max(N,L) by max(K,M) matrix.

y is the integral from x to infinity of the chi-square distribution with n degrees of freedom. The elements of n must all be positive integers. The allowable ranges for the arguments are: x >= 0, n > 0. y = 1-χ(x,n), where χ is the chi-square cdf with n degrees of freedom. -1 is returned for those elements with invalid inputs.

cdffc Computes complement of cdf of F.

Format: y = cdffc(x,n1,n2);

Input: x N×K matrix.

n1 L×M matrix, element by element conformable with x.

n2 P×Q matrix, element by element conformable with x and n1.

Output: y max(N,L,P) by max(K,M,Q) matrix.

y is the integral from x to ∞ of the F distribution with n1 and n2 degrees of freedom. This equals 1 - F(x,n1,n2) where F is the cumulative distribution with n1 and n2 degrees of freedom. Allowable ranges for the arguments are: x ≥ 0, n1 > 0, n2 > 0. -1 is returned for those elements with invalid inputs.

cdfgam Computes the incomplete gamma function. This function is defined as

$$P(x,\text{int lim}) \equiv \frac{1}{\Gamma(x)}\int_0^{\text{int lim}} e^{-t}t^{x-1}dt, \quad \text{for } x > 0 \text{ and intlim } \geq 0.$$[23]

Format: g = cdfgam(x,intlim);

Input: x N×K matrix.

intlim L×M matrix, element by element conformable with xand intlim.

Output: g max(N,L) by max(K,M) matrix.

-1 is returned for those elements with invalid inputs.

cdfn Computes integral of normal distribution: lower tail.

Format: y = cdfn(x);

Input: x N×K matrix.

Output: y N×K matrix.

y is the integral from -∞ to x of the normal density function.

cdfnc Computes complement (1-cdf) of the standard normal distribution.

*** cdfnb** Negative Binomial Cumulative Distribution function

Format: y = cdfnb(x,p,n)

Input: x r × c matrix of number of successes (x >= 0)

p matrix of probability of success (0 < p < 1)

n negative-binomial parameter (n > 0)

(p & n must be element by element conformable with x)

[23] The limiting values of this monotone function are P(x,0) = 0 and P(x,∞) = 1. 1-P(x, intlim) = Γ(x,intlim)/Γ(x) is often also called the incomplete gamma function. The χ^2 cdf is a special case of the incomplete gamma function, prob(χ^2,df) = P(df/2,χ^2/2).

Output: y r × c matrix of probabilities such that. pr(ξ≤ x) = y where ξ ~ Negative-Binomial(p,n)

This uses the relationship between the Negative-Binomial distribution and the Beta distribution. The approximation has an accuracy of 6 decimal places but is more stable for large values of N than using the summation of individual Negative-Binomial terms.

*** cdfp** Poisson Cumulative Distribution function

Format: y = cdfp(x,m)

Input: x r × c matrix of number of successes (x ≥ 0)

m matrix of mean number of successes (m > 0) (m must be element by element conformable with x)

Output: y r × c matrix of probabilities st. pr(ξ≤ x) = y where ξ ~ Poisson(m)

This program uses the relationship between the Poisson distribution and the Chi-squared distribution. (Press, *et al., op. cit.,* p. 165.) The approximation has an accuracy of 8 decimal places but is more stable for large valuesof N than using the summation of individual Poisson terms.

cdftc Computes complement of cdf of t-distribution.

Format: y = cdftc(x,n);

Input: x N×K matrix, Student t deviates

n L×M matrix, degrees of freedom, n > 0. element by element conformable with x.

Output: y max(N,L) by max(K,M) matrix, complementary probability levels.

y is the integral from x to ∞ of the t distribution with n degrees of freedom. y = 1 - t(x,n), where t is the t cdf with n degrees of freedom. The complement of the cdf is computed because this is what is most commonly needed in statistical applications, and because it can be computed with fewer problems of roundoff error. For n <= 0, a -1 is returned.

Multivariate distributions:

cdfbvn Computes the cumulative distribution function of the standardized bivariate normal density (lower tail).

Format: c = cdfbvn(h,k,r);

Input: h N×K matrix, the upper limits of integration for variable 1.

k L×M matrix, element by element conformable with h, the upper limits of integration for variable 2.

r P×Q matrix, element by element conformable with h and k, the correlation coefficients between the two variables.

Output: c max(N,L,P) by max(K,M,Q) matrix

c is the result of the double integral from -∞ to h and -∞ to k of the standardized bivariate normal density. Variable 1 and variable 2 have 0 means, unit variances, and correlation = r. Allowable ranges for the arguments are: -∞ < h < +∞, -∞ < k < +∞, -1 ≤r ≤1.

-1 is returned for those elements with invalid inputs.

cdftvn Computes the cumulative distribution function of the standardized trivariate normal density (lower tail).

Format: c = cdftvn(x1,x2,x3,rho12,rho23,rho31);

Input: x1 N×1 vector of upper limits of integration for variable 1.

x2 N×1 vector of upper limits of integration for variable 2.

x3 N×1 vector of upper limits of integration for variable 3.

rho12 scalar or N×1 vector of correlation coefficients between x1 and x2.

rho23 scalar or N×1 vector of correlation coefficients between x2 and x3.

rho31 scalar or N×1 vector of correlation coefficients between x3 and x1.

Output: c N×1 vector.

c is the result of the triple integral from -∞ to x1, -∞ to x2, and -∞ to x3 of the standardized trivariate normal density. A separate integral is computed for each row of the inputs. rho12, rho23 and rho31 can range from -1 to 1 exclusive, and must come from a legitimate positive definite matrix. -1 is returned for those rows with invalid inputs.

Noncentral distributions:

cdfchinc Computes integral of noncentral chi-square from 0 to x. Can return a vector of values, but the degrees of freedom and noncentrality parameter must be thesame for all values of x.[24]

Format: y = cdfchinc(x,v,d);

Input: x N×1 vector, values of upper limits of integrals, must be greater than 0.
v scalar, degrees of freedom, v > 0.
d scalar, noncentrality parameter, d > 0.
d is the square root of the noncentrality parameter that is sometimes denoted by lambda. See Scheffe, *The Analysis of Variance*, 1959, app. IV.

Output: y N×1 vector, integrals from 0 to x of noncentral chi-square.

cdffnc Computes integral of noncentral F from 0 to x.[25]

Format: y = cdffnc(x,v1,v2,d);

Input: x N×1 vector, values of upper limits of integrals, x > 0.
v1 scalar, degrees of freedom of numerator, v1 > 0.
v2 scalar, degrees of freedom of denominator, v2 > 0.
d scalar, noncentrality parameter, d > 0.
d is the square root of the noncentrality parameter sometimes denoted by lambda.

Output: y N×1 vector of integrals from 0 to x of noncentral F.

cdftnc Computes integral of noncentral t-distribution from -∞ to x. It can return a vector of values,but the degrees of freedom and noncentrality parameter must be the same for all values of x.[26]

---

[24] Technical Notes:
a. Relation to cdfchic: cdfchic(x,v) = 1 - cdfchinc(x,v,0);
b. The formula used by GAUSS is taken from Abramowitz and Stegun, *Handbook of MathematicalFunctions*, National Bureau of Standards, 1972, p. 942, formula 26.4.25.
c. According to GAUSS, the results were verified against the table in *Selected Tables in Mathematical Statistics*, H. L. Harter & D. B. Owen,Markham, 1970, p. 13, for the range of 1 to 100 degrees offreedom, d = 0 to 10 (d*d = 0 to 100 in the table), and alpha (central chi-square) = 0.001 to 0.100. Values outside of these ranges may require an adjustment in TOL.

[25] Technical Notes:
a. Relation to cdffc: cdffc(x,v1,v2) = 1 - cdffnc(x,v1,v2,0);
b. The formula used is taken from Abramowitz and Stegun, *op. cit.*, p. 947, formula 26.6.20

[26] Technical Notes:
a. Relation to cdftc: cdftc(x,v) = 1 - cdftnc(x,v,0);
b. The formula used by GAUSS is based on the formula in SUGISupplemental Library User's Guide, 1983, SAS Institute, page 232 (which is attributed to Johnson and Kotz, 1970). The formula used by GAUSS is a modification of that formula. It has been tested against direct numerical integration, and against

Format: y = cdftnc(x,v,d);
Input: x N×1 vector, values of upper limits of integrals.
v scalar, degrees of freedom, v > 0.
d scalar, noncentrality parameter. d is the square root of the noncentrality parameter that is designated by lambda.
Output: y N×1 vector, integrals from -infinity to x of noncentral t.

Inverse distributions[27]:

cdfni Computes inverse of cdf of normal distribution.
Format: x = cdfni(p);
Input: p N×K real matrix, normal probability levels, 0 ≤ p ≤ 1
Output: x N×K real matrix, normal deviates, such that cdfn(x) equals p

*** invcdfb** Inverse Binomial Cumulative Distribution function
Format: x = invcdfb(y,p,n)
Input: y r × c matrix of probabilities (0 < y < 1)
p matrix of probability of success (0 < p < 1)
n matrix of number of trials (n > 0)
(p & n must be element by element conformable with x)
Output: x r × c matrix of number of successes (0 < x < n) such that pr(ξ≤ x) = y, where ξ ~ binomial(p,n)
This program uses the relationship between the Binomial distribution and the F distribution. The approximation has an accuracy of 6 decimal places but is more stable for large values of N thanusing the summation of individual Binomial terms.

*** invcdfp** Inverse Poisson Cumulative Distribution function
Format: x = invcdfp(y,m)
Input: y r × c matrix of probabilities (0 < y < 1)
m matrix of mean number of successes (m > 0)
(m must be element by elemnet conformable with x)
Output: x r × c matrix of number of successes (x ≥ 0) such that Pr(ξ ≤ x) = y, where ξ ~ Poisson(m)
This program uses the relationship between the Poisson distributionand the Chi-squared distribution. (Press, *et al., op. cit.,* p. 165.) The approximation has anaccuracy of 8 decimal places but is more stable for large values of N than using the summation of individual Poisson terms.

*** invcdfnb** Inverse Negative-Binomial Cumulative Distribution function
Format: x = invcdfnb(y,p,n)
Input: y r × c matrix of probabilities (0 < y < 1)
p matrix of probability of success (0 < p < 1)
n negative-binomial parameter (n > 0)
(p & s must be eelement by elemente conformable with x)
Output: x r × c matrix of number of successes (x ≥ 0) such that Pr(ξ≤x) = y, where ξ ~ Negative-Binomial(p,n)

---

simulation experiments in which noncentral t randomvariates were generated and the cdf found directly.

[27] The problem solved by the inverse distribution is this: I give you a probability p, i.e., a number between 0 and 1, you tell me the value of x which makes $invdist(x) = \int_{-\infty}^{x} dist(x)dx = p$. This has obvious application to the problem of generating a sequence of random variables from a particular distribution knowing only a sequence of RVs uniformly distributed on the interval (0,1). However, it will not often be the most efficient method of doing so, nor will we always know what the closed form solution to the above is, i.e., what inverse distribution corresponds to the distribution we want to investigate.

This program uses the relationship between the Negative-Binomial distribution and the Beta distribution.[28] The approximation has an accuracy of 6 decimal places but is more stable for large values of N than using the summation of individual Negative-Binomial terms.

cdftci Computes inverse of complement of t-distribution cdf.

Format: x = cdftci(p,n);

Input: p N×K real matrix, complementary Student's t probability levels, $0 \le p \le 1$

n L×M real matrix, degrees of freedom, $n \ge 1$, n need not be integral. element by element conformable with p.

Output: x max(N,L) by max(K,M) real matrix, Student's t deviates, such that cdftc(x,n) equals p.

*** invcdff** Inverse of the F Cumulative Distribution function with n1,n2 degrees of freedom

Format: x = invcdff(p,n1,n2)

Input: p matrix of percentage points (0 < p < 1)

n1 matrix of numerator df (conformable with p)

n2 matrix of denominator df (conformable with p)

Output: x matrix of critical values st Pr(ξ < x) = p and ξ ~F(n1,n2).

*** invcfchi** Inverse of the Chi squared Cumulative Distribution function with n1 degrees of freedom

Format: x = invcfchi(p,n)

Input: p matrix of percentage points (0 < p < 1)

n matrix of degrees of freedom (n > 0) (conformablewith x)

Output: x matrix of critical values st Pr(ξ< x) = p and ξ ~ Chi(n)

*** invcdfg** Inverse of the Gamma Cumulative Distribution function

Format: x = invcdfg(p,a)

Input: p matrix of percentage points (0 < p < 1)

a matrix of shape parameters (conformable with p)

Output: x matrix of critical values st pr(ξ < x) = p and ξ ~ gamma(a)

Basic18.gss illustrates the use of some of the GAUSS commands and David Baird's procedures for generating random numbers and comparing the empirical distributions of these with the theoretical densities and /or cumulative distribution functions.

15. NUMERICAL DIFFERENTIATION AND INTEGRATION

GAUSS has a number of built-in procedures for doing numerical differentiation and integration.

a. Numerical Differentiation

gradp and **hessp** use a finite difference approximation.[29]

---

[28] N. L. Johnson, S. Kotz and A. W. Kemp, *Univariate Discrete Distributions*, 2nd edition.New York: Wiley, 1992, p. 206.

[29] There is a large literature in numerical analysis on interpolation formulae, a topic which is closly related to to the problem of finding the numerical value of the derivative of a function or its gradient at a particular value of its argument (in the case of more than one argument, the gradient is a vector; when the "function"

gradp Computes first derivative of a function or the gradient vector or matrix (Jacobian) of a vector-valued function that has been defined in a procedure.Single-sided (forward difference) gradients are computed.

Format: g = gradp(&f,x0);

Input: f scalar, procedure pointer to a vector-valued function:

$$f: K\times 1 \rightarrow N\times 1$$

The argument of f must be a single vector so if there are other arguments or data they must be defined globally. f need not return a scalar but it must return a single vector. This is very common, for example, when f is a likelihood function: the argument of f is a vector of parameters but the data also enter as globals. The output may be a single scalar value for all the observations or may be a vector with dimension equal the number of observations, the likelihood of the parameter values at each observation being returned.

x0 K×1 vector of points at which to compute gradient.

Output: g N×K matrix containing the gradients of f with respect to the variable x at x0.

gradp will return a row for every row that is returned by f. For instance, if f returns a 1×1 result, then gradp will return a 1×K row vector. This allows the same function to be used where N is the number of rows in the result returned by f. Thus, for instance, **gradp** can be used to compute theJacobian matrix of a set of equations.There is also a "bug" in **gradp** which is of no practical consequence: **gradp** will often return exactly 0 rather than a very small number, e.g. 1e-08; this can be disconcerting since only integer values can ever be exactly zero inside a computer.

There are three additional gradient routines which may be called directly in a GAUSS program which are not documented in the online help menu or in the manual:

gradre, Computes the gradient vector or matrix (Jacobian) of a vector-valued function that has been defined in a procedure. Single-sided (forward difference) gradients are computed, using Richardson Extrapolation (see Young and Gegory, *op. cit.*, pp. 1067 - 1068). The algorithm, is an iterative process which updates a derivative based on values calculated in a previous iteration. This is slower than GRADP, but can in general, return values that are accurate to about 8 digits of precision. The algorithm runs through n iterations. _n is a global whose default is 25. Format, Input, and Output art the same as for **gradp.**

Variants are:

gradcd Central difference numerical gradients.

gradfd Forward difference numerical gradients.

For more information, check the **gradient.src** file.

---

has more than one output, the derivative is a matrix called the *Jacobian*). See Press, *et al., op. cit.,* pp. 77 - 89; Stoer and Bulirsch, *op. cit.*,pp. 37 - 124; A. Ralston, *A First course in Numerical Analysis,* New York: McGraw- Hill, 1965; and D. M. Young and R. T. Gregory, *A Survey of Numerical Mathematics, Vol. 1,* New York: Dover Publications, 1973, pp. 344 - 361. In both the case of numerical differentiation and in the case of numerical integration, which is also called *quadrature*, the function to be differentiated, say f(x), is replaced by another function, say F(x), for which it is possible to obtain analytical derivatives which are computationally tractable. Often F(x), for example, is a polynomial which agrees with f(x) at a certain number of *interpolation points* and certain of whose derivatives agree with those of f(x) at the corresponding points. Surprisingly, such numerical evaluation of derivatives can be more accurate at the limited number of points selected than straight evaluation from the general analytical derivatives. Both **gradp** and **hessp** use a finite difference approximation based on two points; see young and Gegrory, *op. cit.,* pp. 350 -354. This can be quite fast but may be inaccurate for rapidly changing functions. See the code in **gradp.src** and **hessp.src**.

hessp Computes the matrix of second partial derivatives (Hessian matrix) of a function defined by a procedure.

Format: h = hessp(&f, x0);

Inputs: &f pointer to a single-valued function f(x), defined as a procedure, taking a single Kx1 vector argument (f:K×1 → 1×1). It is acceptable for f(x) to have been defined in terms of global arguments in addition to x.

x0 Kx1 vector specifying the point at which the Hessian of f(x) is to be computed.

Output: h KxK matrix of second derivatives of f with respect to x at x0. This matrix will be symmetric.

This procedure requires K(K+1)/2 function evaluations. Thus if K is large it may take a long time to compute the Hessian matrix.No more than 3 - 4 digit accuracy should be expected fromthis function, though it is possible for greater accuracy to be achieved with some functions.It is important that the function be properly scaled, in order to obtain greatest possible accuracy. Specifically, scale it so that the first derivatives are approximately the same size. If these derivatives differ by more than a factor of 100 or so, the results can be meaningless.

b. Numerical Integration[30]

intquad1, **intquad2**, and **intquad3** use Gaussian quadrature to calculate the integr the user-defined function over a rectangular region.To calculate an integral over a region defined by functions of x and y,use **intgrat2** and **intgrat3**.To get a greater degree of accuracy than that provided by **intquad1**, use **intsimp** for one-dimensional integration.

intquad1 Integrates a 1-dimensional function using Gauss-Legendre quadrature (Young and Gregory, *op. cit.*, pp 401 - 412). A number of upper and lower bounds may be calculated in one procedure call.

Format: y = intquad1(&f,xl);

Input: &f pointer to the procedure containing the function to be integrated. The function f must be capable of function values. intquad1 will pass to f a vector, and expect a matrix in return.

xl 2×N matrix, the limits of x.The first row is the upper limit and the second row is the lower limit. N integrals are computed.

_intord scalar, the order of the integration. The larger **_intord** the more precise the result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. The default is 12.

Output: y Nx1 vector of the estimated integral(s) of f(x) evaluated between between the limits given by xl.

intquad2 Integrates a 2-dimensional function. Same format as **intquad1**. User defined function f must return a vector of function values. intquad2 will pass to user defined functions a vector or matrix for x and y and expect a vector or matrix to be returned. Use the ".*" and "./" instead of "*" and "/". intquad2 will expand scalars to the appropriate size. This means that functions can be defined to return a scalar constant. If users write their functions incorrectly (using "*" instead of ".*", for example), intquad2 will not compute the expected integral, but

[30] This is a big subject: See Young and Gregory, *op. cit.*, pp. 361 - 421; Press, *et al., op. cit.*, pp.

the integral of a constant function. To integrate over a region which is bounded by functions, rather than just scalars, use **intgrat2**.
IMPORTANT: There is another global which must be set
_intrec scalar. This determines the recursion level. Users should always explicitly set this to 0 in their command files before calling **intquad2**, because this variable is used to keep track of the lvel of recursion of **intquad2** and may start out with a different value in your program if **intquad2** has previously been called.

intquad3 Integrates a 3-dimensional function. Same format and instructions as **intquad2**.

intsimp Integrates a specified function using Simpson's method with end correction (see Young and Gregory, *op. cit.*, pp. 365 - 368). A single integral is computed in one function call. This is more accurate than **intquad1** but slower.

Format: y = intsimp(&f,xl,tol);

Inputs:
- &f pointer to the procedure containing the function to be integrated. The function f must be capable of function values. **intsimp** will pass to f a vector, and expect a vector in return. Use ".*" instead of "*".
- xl 2×1 vector, the limits of x.The first element is the upper limit and the second element is the lower limit.
- tol scalar, tolerance to be used in testing for convergence.

Output: y scalar, the estimated integral of f(x) between x[1] and x[2].

intgrat2 Integrates over a region defined by functions of x. Evaluates a the double integral

$$\int_a^b \int_{g_2(x)}^{g_1(x)} f(x,y)\,dx\,dy,$$

where g_1, g_2, and f are user defined functions and a and b are scalars.

Format: y = intgrat2(&f,xl,gl);

Input:
- &f pointer to the procedure containing the function to be integrated.
- xl 2×N matrix, the limits of x. These must be scalar limits.
- gl 2×N matrix of function pointers, the limits of y.

For xl and gl, the first row is the upper limit and the second row is the lower limit. N integrals are computed.User defined functions f, and those used in gl must either

1. Return a scalar constant or,
2. Return a vector of function values.

intgrat2 will pass to user defined functions a vector or matrix for x and y and expect a vector or matrix to be returned.

_intord scalar, the order of the integration. The larger _intord, the more precise the final result will be. _intord may be set to:

2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40

Default is 32.

_intrec scalar. This determines the recursion level. Users should always explicitly set this to 0 in their command files before calling **intgrat2.**

Output: y Nx1 vector of the estimated integral(s) of f(x,y) evaluated between between the limits given by xl and gl.

intgrat3 Evaluates the triple integral

$$\int_a^b \int_{g_2}^{g_1} \int_{h_2}^{h_1} f(x,y,z)dzdydx ,$$

where h_1, h_2, g_1, g_2, and f are user defined functions and a and b are scalars.
Same format, input and output as **ingrat2**.

basic19.gss illustrates the use of **gradp, intquad1 inquad2, and ingrat1** for univariate and bivariate normal distributions.

One of the most important uses of **gradp** and **hessp** is to obtain asymptotic standard errors for maximum-likelihood estimates. This is illustrated in **basic20.gss** for the ship accident data analysed in the next section of thse notes. In the next section of these Notes, I estimate the Poisson count model applied in Greene (1993, pp. 676 - 679) to a study of wave damage to cargo ships.[31] The data originally used by Lloyd's to set insurance premia for cargo ships, were collected to calculate the risk of damage associated with three factors: (1) ship type; (2) year of construction; and (3) period of operation. "It seems reasonable to suppose that the number of damage incidents is directly proportional to the aggregate months of service or total period of risk." (McCullagh and Nelder, *op. cit.,* p. 206.) The problem of running a regression for this example is that the number of accidents experienced for some classes of vessels and some years of construction are zero and some are very small. As Greene suggests, a Poisson model may be more appropriate.[32] The object of the analysis is to see whether any of the factors other than service or exposure to risk, affect the probability of damage. The analysis attempts to assess the probability of accident as a function of ship type, year of construction, and exposure. The first two are represented by dummies and the last by continuous variable which varies by a factor of more than 450 from its lowest to its highest value. The probability of an accident in a unit period is assumed to be

$$\text{Prob}(Y_i = y_i) = \frac{e^{-l_i} l_i^{y_i}}{y_i !,}, \quad y_i = 1,2,..., \quad l_i > 0.$$

$$\ln l_i = b'x_i .$$

To estimate the parameters β, I eliminate ship type A and construction year 60 - 64 and add an overall constant. The results obtained using the method of steepest ascent are computed for Greene's data with the last explanatory variable, period of operation, rescaled by a factor of 1000:

Steepest Ascent Method applied with start value =
1 1 1 1 1 1 1 1 0
and maxiter = 100

Gradient at start values =
-78.47 76.35 -62.65 -52.65 -47.65 -22.73 -0.7312 -52.73 1208
iter = 7

[31] The data are from P. McCullagh and J. A. Nelder, *Generalized Linear Models,* 2nd. ed., London: Chapman & Hall, 1989, p.205.
[32] See Hogg and Craig, *op. cit.*, pp. 99 - 101, for a good discussion of the rationale for using the Poisson model to study count data.

```
Maximizing value of beta =
    1.121     -0.4321     -1.895     -0.7933     -0.5191      0.4029      1.411      0.9141      0.1426
 logLikelihood at the maximum =     -46.8
Gradient at the maximum =
-5.704e-006  9.867e-006 -2.625e-006 -8.957e-007 -1.369e-006 -5.29e-006  -2.518e-006 -1.555e-006 -4.984e-006
Standard errors implied by the Hessian evaluated at the maximum:
   0.2742      0.4404      0.4802      0.3114      0.2838      0.2392      0.2281      0.2921      0.03074
Assymptotic t-values implied by the maximum:
    4.089     -0.9811     -3.945      -2.547      -1.829       1.684       6.186       3.129       4.638
Runtime method of steepest ascent       0.08              seconds.
```

These are essentially the same as those reported by Greene, p. 676.

basic20.gss reruns the steepest ascent for this problem to illustrate the use of GRADP and HESSP in the maximization and their use in obtaining the asymptotic standard errors and t-statistics from the likelihood function, both directly from the hessian evaluated at the maximum and also from the gradient using the outer product method. The latter yields:

```
Standard errors implied by the Hessian evaluated by the outer product method:
   0.2522      0.6401       1.368      0.3098      0.3399      0.3294      0.2052      0.3771      0.04294
Assymptotic t-values implied by the maximum:
    4.445      -0.675      -1.385       -2.56      -1.527       1.223       6.876       2.424        3.32
```

which are not too far from the results presented above, obtained directly from HESSP, although some differ enough to make a difference in the conventional assessment of significance. In the next section I estimate some examples in which I also calculate analytical hessians; I find that the results differ hardly at all from HESSP, but both differ somewhat from those obtained using the outer product method.

16. MAXIMAZATION AND MINIMIZATION

As the last example illustrates, it is often very easy to write your own GAUSS procedure to mazximize a function you have specified -- provided you know what you are doing and the function you are working with is sufficiently well-behaved. The advantage of doing so is two-fold: You are forced to think carefully about the problem you are trying to solve and to write it down as a maximization or a minimization problem. Second, you are not dealing with a "black box"; you know what goes in and, because you know what you've written does, you know what comes out and how to interpret it. Although there is really no substitute for knowing what you are doing, there are several complications involved in writing your own completely from scratch: The function particular to your problem may involve a great many parameters; many dimensions greatly increases the complexity of the task. Concommitantly, your function may *not* be well-behaved, a possibility which is enhanced by high dimensionality of the parameter space. Moreover, having a lot of parameters around may make it difficult for you to understand the "wrinkles" in the function with which you are working. When you are dealing with a small number of parameters, it is possible to use grid search and graphical techniques to better understand the nature of the function, and even to get a pretty good idea of where the global maximum or minimum is.[33] But when the number of parameters is large, and especially if there are constraints (of equality or inequality) invloved, more sophisticated procedures may be necessary. Efficiency may also be a problem; brute force may take far too long to achieve a result.

Press, *et al., op. cit.,* pp.274 - 327, Chapter 10 (minus section 9), give a good discussion of all the various techniques involved including the simplex method for linear programming. Except for the simplex method, Bo Honore has programmed all of the algorithms discussed by Press, *et al.,op. cit.,* in GAUSS. His procedures are available on the Web at:

http://webware.princeton.edu/econometrics/programs/gaussmax/optimum.prc

The simplex method has been programmed by Thierry Roncalli in GAUSS and is avalable from his Web site at:

http://www.montesquieu.u-bordeaux.fr/u/roncalli/gauss.html

in the section labeled "Data Envelopment Analysis."

---

[33] In a recent paper, available on the Web, "A Toolkit for Optimizing Functions in Economics," June 1997, Bill Goff, suggests a number of graphical techniques and measures which are helpful in understanding the nature of the function and what more sophisticated techniques might be profitably employed.

In addition to two on-line procedures:

qnewton which uses the Broyden-Fletcher-Goldfarb-Shanno (BFGS) descent algorithm[34] to minimize a user supplied function; and

qprog which solves the standard quadratic programming problem:

minimize 0.5 * x'Qx - x'R
subject to constraints,
Ax = B
Cx >= D
and bounds to the maximum and minimum values of the individual elements of the vector x,

GAUSS itself has an extensive library of routines designed to maximize likelihood functions and to minimize functions subject to constraints. These are available (at an extra charge!) in the form of five easily installed applications modules:

1. Maximum Likelihood: procedure **maxlik** and related procedures. An older version of Constrained Maximum Likelihood (**CML**) without the ability to impose constraints.

2. Optimization: procedure **optimum** and related procedures. An older version of Constrained Optimization (**CO**) without the ability to impose constraints.

3. Constrained Maximum Likelihood (**CML**): procedure **cml** and related procedures for bootstrapping and for count and duration data. The later are written by Gary King and are extensively used and described in his book, *Unifying Political Methodology: The Likelihood Theory of Statistical Inference*, Cambridge: University Press, 1989. Updated versions of these programs are available at his Web site:

http://data.fas.harvard.edu/gov-dept/

This set pof procedures will maximize a user supplied likelihood function subject to linear or nonlinear equality and inequality constraints:

$$\max L = \sum_{i=1}^{N} \log P(Y_i; q)^{w_i}$$

where N is the number of observations, $P(Y_i;\theta)$ is the probability of Y_i given θ, a vector of parameters subject to the following constraints:

linear constraints
$A\theta = B$
$C\theta \geq D$
nonlinear constraints
$G(\theta) = 0$
$H(\theta) \geq 0$
and bounds
$\theta_L \leq \theta \leq \theta_U$.

4. Constrained Optimization (**CO**): a set of procedures for the solution of the general nonlinear programming problem:

min $F(\theta)$
subject to the linear constraints
$A\theta = B$
$C\theta \geq D$
the nonlinear constraints
$G(\theta) = 0$
$H(\theta) \geq 0$
and bounds
$\theta_L \leq \theta \leq \theta_U$.

5. Linear Programming: **simplex** and related procedures for solving the small-scale linear programming problem:

[34] Press, *et al., op.cit.,* pp. 308 - 312.

$$\max \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to } \sum_{j=1}^{n} a_j x_j \le b_i \quad (i = 1,\ldots,m)$$

$$l_j \le x_j \le u_j \quad (j = 1,\ldots,n).$$

I will deal with the use of **CML**, **CO,** and **SIMPLEX** in a separate tutorial: "Optimization in GAUSS: A Tutorial," as well as with how to use grid search optimiztion procedures. In closing this tutorial, I simply illustrate the use of **QNewton** and **QProg** as well as the method of steepest ascent used in **basic20.gss** in a series of five examples**.**

a. Examples Comparing Steepest Ascent with QNewton

QNewton Minimizes a user supplied function using the BFGS descent algorithm

Format: { x,f,g,retcode } = QNewton(&fct,x0)

Input:
- &fct — pointer to a procedure that computes the function to be minimized. This procedure must have one input argument, a vector of parameter values, and one output argument, the value of the function evaluated at the input vector of parameter values.
- x0 — K×1 vector of start values

Output:
- x — K×1 vector of parameters at minimum
- f — scalar function evaluated at x
- g — K×1 gradient evaluated at x
- retcode — return code:
  - 0 normal convergence
  - 1 forced exit
  - 2 maximum number of iterations exceeded
  - 3 function calculation failed
  - 4 gradient calculation failed
  - 5 step length calculation failed
  - 6 function cannot be evaluated at initial parameter values

Globals:
- _qn_RelGradTol — scalar, convergence tolerance for relative gradient of estimated coefficients. Default = 1e-5.
- _qn_GradProc — scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. This procedure must have a single input argument, a Kx1 vector of parameter values, and a single output argument, a Kx1 vector of gradients of the function with respect to the parameters evaluated at the vector of parameter values.
- _qn_MaxIters — scalar, maximum number of iterations. Default = 1e+5.
- _qn_PrintIters — scalar, if 1, print iteration information. Default = 0.
- _qn_ParNames — K×1 vector, labels for parameters
- _qn_RandRadius — scalar, If zero, no random search is attempted. If nonzero it is the radius of random search which is invoked whenever the usual line search fails. Default = .01.
- __output — scalar, if 1, prints results.

QNewton is recursive, that is, it can call a version of itself with another function and set of global variables.

In **basic21.gss** through **basic25.gss**, I present five examples of maximum likelihood estimation: (1) the Poisson regression example discussed in the previous section; (2) an example involving regression with first-order serially correlated disturbances; (3) an example involving regression with heteroskedastistic disturbances; (4) estimation of a regression with a Box-Cox transformation; and (5) nonlinear regression. All of the programs are reproduced in the appendix and are also available at the tutorial Web site.

(1) *Poisson Regression*: The example has been described in the previous section. The program to compare steepest ascent and QNewton and graph the likelihood function in the vicinity of the maximum is **basic21.gss.**

The result of running this program are as follows (he 9 graphs are reproducedin the appendix):

```
This program runs a Poisson Regression for the
ship data given in Greene, 2nd ed.,Table 21.12, p. 676, to test the speed of convergence of
the method of steepest ascent versus QNewton procedures.  Graphs of the likelihood function
in the vicinity of the maximizing values are given. Ship service is rescaled by a factor of 10^-3.
basic21.gss. Last updated:    4/14/98  at  12:17:45

-------------------------------------------------------------------------------------------------------------------------------
Steepest Ascent Method applied with start value =     1    1    1    1    1    1    1    1      0
and maxiter =    100
Gradient at start values =
-78.47       76.35       -62.65       -52.65       -47.65       -22.73       -0.7312      -52.73       1.208e+006
iter =     8
Maximizing value of beta =
    1.121       -0.432       -1.895       -0.7933       -0.5191       0.4029       1.411       0.9141    0.0001426
 logLikelihood at the maximum =      -46.8
Gradient at the maximum =
6.338e-006  -9.868e-006   0  -1.791e-006    -1.369e-006  1.411e-005    0   2.332e-006     -0.0003553
Standard errors implied by the Hessian evaluated at the maximum:
   0.2742       0.4395       0.4802       0.3114       0.2837       0.2392       0.2281       0.2921   3.068e-005
Assymptotic t-values implied by the maximum:
    4.089       -0.983       -3.946       -2.548       -1.83         1.684        6.187        3.13         4.647
Runtime method of steepest ascent        0.08                seconds.
-------------------------------------------------------------------------------------------------------------------------------
There is a problem with b9: The likelihood function changes too rapidly in the vicinity of
the maximum! Here are some values around the maximizing value: (Remove /*  */ to see example).
QNewton has a problem with such dramatic variation. The function calculation will fail if b9
deviates too far from the maximizing value due to overflow.
-------------------------------------------------------------------------------------------------------------------------------
QNewton applied with rescaled data[11,.] = x[9,.]:
Start =    1   1            1            1            1            1            1            1            0            Maxiter =   50
Return code =     0
Value of log likelihood =      -46.8
beta' =      1.121          -0.4321      -1.895       -0.7933      -0.5191       0.4029       1.411        0.9141       0.1426
gradient = -4.437e-006                    0         1.5e-006          0       -1.369e-006                3.527e-006                -3.022e-006
               7.773e-007                5.981e-005
Standard errors implied by the Hessian evaluated at the maximum:
   0.2742       0.4403       0.4802       0.3114       0.2838       0.2392       0.2281       0.2921       0.03074
Assymptotic t-values implied by the maximum:
    4.089       -0.9813      -3.945       -2.547       -1.829        1.684        6.186        3.129        4.638
Runtime for QNewton =       0.09                seconds.
_______________________________________________________________________________
Runtime = 0.81 seconds
```

(2) *First-order residual autocorrelation.* The log likelihood function for the case of first-order residual autoregression is

$$(1)\qquad L(b, r, s^2 \mid y_t, x_t, t=1,\ldots,T) =$$
$$-\frac{\mathrm{T}}{2}\log 2\pi - \frac{\mathrm{T}}{2}\log\sigma^2 + \frac{1}{2}\log(1-\rho^2)$$
$$-\frac{1}{2\sigma^2}\left\{(1-\rho^2)(y_1 - x_1'\beta)^2 + \sum_{t=2}^{\mathrm{T}}\left[(y_t - x_t'\beta) - \rho(y_{t-1} - x_{t-1}'\beta)\right]^2\right\}.$$

Holding ρ fixed at some value in the interval (-1, 1) shows that β(ρ) are just the GLS estimates but $\sigma^2(\rho)$ is a slight modification:

$$(2)\qquad \hat{s}^2(r) = \frac{1}{T}\left\{(1-r^2)(y_1 - x_1'\hat{b}(r))^2 + \sum_{t=2}^{T}\left[(y_t - x_t'\hat{b}(r)) - r(y_{t-1} - x_{t-1}'\hat{b}(r))\right]^2\right\}.$$

Call these values β(ρ) and $\sigma^2(\rho)$ *as functions of r*. To concentrate the support function, substitute in

$$(3)\qquad L^*(r \mid y_t, x_t, t=1,\ldots,T)$$

$$= -\frac{T}{2}\log 2p - \frac{T}{2}\log \hat{s}^2(r) + \frac{1}{2}\log(1-r^2) - \frac{1}{2s^2(r)}\cdot T\cdot \hat{s}^2(r)$$

$$= -\frac{T}{2}(\log 2p + 1) - \frac{T}{2}\log \hat{s}^2(r) + \frac{1}{2}\log(1-r^2).$$

This shows that the ML estimates of ρ and of β and σ^2 are obtained by minimizing

$$T\log \hat{s}^2(r) - \log(1-r^2)$$

where $\hat{s}^2(r)$ is the estimate of σ^2, from(2). Of course, this function is determined numerically by the least-squares procedure.

Note, that since the term with T dominates for large T, this also shows that the *asymptotic* ML estimates can be obtained by iterating on ρ to obtain the smallest GLS estimate of σ^2. But for small samples the term $(1 - \rho^2)$ is more important and the iterated GLS (sometimes called Yule-Walker) will not be the same approximately as the ML estimates.

In this simple case $L^*(\rho)$ is a function of only one parameter which is a priori constrained to lie in the interval (-1, 1), so a *grid search* is feasible and easy.

What happens if the maximum occurs on a boundary point, ρ = -1 or 1? At such a point any quadratic approximation is clearly invalid.

The information matrix can be obtained by differentiating the original likelihood function with respect to β, σ^2 *and* ρ. It is *not* block diagonal (as in the standard regression case for β and σ^2) and therefore the correctly estimated variance-covariance matrix for β is *not the same as the GLS estimate for the maximizing value of r*.

Judge, et al. (1988, pp. 405-409) consider an artificially constructed example of a regression with two independent variables and a first-order serially correlated disturbance (the true values of ρ and the other parameters are not reported):

$$y = x_1 b_1 + x_2 b_2 + x_3 b_3 + u, \text{ where } x_1 = (1,1,1,\ldots,1)' \text{ and } u_t = r\,u_{t-1} + e_t,\ e_t \overset{iid}{\sim} n(0, s^2).$$

There are only 20 observations, which is relatively short for time-series analysis. The data are plotted by **basic23.gss**. The program and results are given in an appendix to these notes.I have used N in my programrather than T. Note that both the full and the concentrated likelihood functions contain singularities at ρ = 1 and -1 of +∞; hence, if you graph these functions, using a finite precision machine, near these values of ρ you will obtain false local minima which you should ignore.

The results for maximization of the concentrated likelihood function from a run of steepest ascent and QNewton are:

Steepest Ascent Method applied with start value = 0.3
and maxiter = 100
Gradient at start value(s) = 7.4504645
iter = 5
Maximizing value of rho = 0.62686887
Log Concentrated Likelihood at the maximum = -46.230742
Gradient Concentrated Likelihood at the maximum = -1.1334791e-006
Hessian Concentrated Likelihood at the maximum = -25.221068

Runtime for method of steepest ascent = 0.01 seconds.

Estimates of remaining parameters derived from transformed OLS regression:
beta' = 4.048 1.661 0.7698 sigma2 = 6.111
Value of the Full Likelihood Function at the Maximum = -46.73
Standard errors implied by the Hessian evaluated at the maximum:
6.352 0.3002 0.1412 0.1766 1.937
Assymptotic t-values implied by the maximum:
0.6373 5.534 5.451 3.551 3.154

QNewton applied with start value = 0.3

==========

QNewton Version 3.2.29 4/19/98 4:53 pm

==========

```
return code =    0
normal convergence

Value of objective function      46.230742

Parameters    Estimates    Gradient
------------------------------------------
rho             0.6269      -0.0003
Number of iterations       4
Minutes to convergence     0.00050
Maximizing value of rho =       0.62685514
Log Likelihood at the maximum =       -46.230742
Gradient at the maximum =    -0.0003491192
Hessian at the maximum =       -25.223653
Return code =    0
Runtime for QNewton = 0.03   seconds.
-----------------------------------------------------------------------------------------------------------------------------
```

Note that they are identical but thet steepest ascent takes one-third of the time. QNewton is a minimization routine so I *minimize the negative of the likelihood function.*

The results obtained from applied both methods to the full likelihood function (in all four parameters without concentration) are:

```
-----------------------------------------------------------------------------------------------------------------------------
Method of steepest ascent applied to the full likelihood:
Steepest Ascent Method applied with start value =    4.5    1.6   0.75    0.5      6
and maxiter =    100
Gradient at start value(s) =          1.22576          26.6137                26.4924                5.07608                0.187856
iter =      6
Maximizing value of parameters =          4.66717             1.64428             0.761625             0.562113          6.10294
Log Likelihood at the maximum =         -46.6563
Gradient Likelihood at the maximum =    -1.52243e-007          -8.64262e-007          -9.32929e-007       0     -2.32853e-007
Hessian Likelihood at the maximum =
  -7.090118e-001       -1.483805e+001       -1.490675e+001        1.551097e-003        6.803060e-006
  -1.483805e+001       -3.247555e+002       -3.203757e+002       -1.370703e+000        2.703405e-004
  -1.490675e+001       -3.203757e+002       -3.670100e+002        5.511979e+000        1.250655e-004
   1.551097e-003       -1.370703e+000        5.511979e+000       -3.213763e+001       -1.346605e-001
   6.803060e-006        2.703405e-004        1.250655e-004       -1.346605e-001       -2.684740e-001

Runtime method of steepest ascent =   0.06   seconds.
Maximizing value of parameters =          4.66717             1.64428             0.761625          0.56211    6.10294
Standard errors implied by the Hessian evaluated at the maximum:
        5.81048              0.280103             0.145275              0.179278              1.93206
Assymptotic t-values implied by the maximum:
       0.803233              5.87025              5.24264              3.13542              3.15878
-----------------------------------------------------------------------------------------------------------------------------
QNewton applied to the Full Likelihood Function:
QNewton applied with start value =              4                   1.6                    0.75                     0.6                      6

================================================================================
 QNewton Version 3.2.29                          4/19/98   4:53 pm
================================================================================

return code =    0
normal convergence

Value of objective function      46.656350

Parameters    Estimates    Gradient
------------------------------------------
beta1          4.6696       0.0001
beta2          1.6442       0.0002
beta3          0.7616       0.0003
rho            0.5621       0.0003
sigma2         6.1028      -0.0000

Number of iterations      24
Minutes to convergence     0.00233
Maximizing value of para =       4.6696347    1.644173    0.7616156   0.56212664     6.1027607
Log Likelihood at the maximum =       -46.65635
```

Gradient at the maximum = 8.429796e-005 2.411441e-004 3.041389e-004 3.235907e-004 -4.540759e-005
Hessian at the maximum =

-7.090008e-001	**-1.483767e+001**	**-1.490631e+001**	**1.919345e-003**	**2.719866e-005**
-1.483767e+001	**-3.247472e+002**	**-3.203671e+002**	**-1.366771e+000**	**3.283005e-004**
-1.490631e+001	**-3.203671e+002**	**-3.670028e+002**	**5.517801e+000**	**2.501415e-004**
1.919345e-003	**-1.366771e+000**	**5.517801e+000**	**-3.213975e+001**	**-1.345482e-001**
2.719866e-005	**3.283005e-004**	**2.501415e-004**	**-1.345482e-001**	**-2.685104e-001**

Return code = 0
Runtime for QNewton = 0.16 seconds.

The results are the same for the two methods but slightly different than the results obtained from the concentrated likelihood function; the problem is the small difference in the value of determined by maximizing the concentrated likelihood function.

(3) *Heteroskedasticity*. Consider a regression in which the residual variance is a function of an exogenous variable z_t. If z_t is a dummy variable, it simplifies matters considerably since, essentially, it permits us to divide the observations into *two* groups, each with a different but constant variance. But let us consider the more general problem here: Let

(4) $s_t^2 = \exp(\gamma_0 + \gamma_1 z_t)$.

The problem is to find γ_0 and γ_1 *and* β and σ^2 in

(5) $y_t = x_t'\beta + u_t$

$Eu_t = 0, \quad Eu_t u_t' = \sigma_t^2, \quad t = t'$

$= 0$, otherwise.

The density of $(y_t | x_t)$ t = 1, ..., T is given by

(6) $$f(y_1,\ldots,y_T|x_1,\ldots,x_T) = \left(\frac{1}{\sqrt{2p}}\right)^T |\Omega^{-1}|^{\frac{1}{2}}\ e^{-(1/2)(y-Xb)'\Omega^{-1}(y-Xb)}$$

where

$$\Omega = \begin{bmatrix} \exp(\gamma_0 + \gamma_1 z_1) & 0 & \cdots & 0 \\ 0 & \exp(\gamma_0 + \gamma_1 z_2) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \exp(\gamma_0 + \gamma_1 z_T) \end{bmatrix}$$

so that

$$\left|\Omega^{-1}\right|^{1/2} = \prod_{t=1}^{T} \frac{1}{\sqrt{\exp(\gamma_0 + \gamma_1 z_t)}}\ .$$

Note: When maximizing numerically it pays to ensure that the variance s_t^2 (and standard deviation) will always be positive (real). This is always the case in this example because the exponential function is never ≤ 0, but other functions might be used in practice for which this could be a problem.

Thus, the log-likelihood function is

(7) $$L(b, \gamma_0, \gamma_1 | y, X) = \frac{T}{2}\log 2p - \frac{1}{2}\sum_{t=0}^{T}(\gamma_0 + \gamma_1 z_t) \quad - \frac{1}{2}\sum_{t=1}^{T}\frac{(y_t - x_t'b)^2}{\exp(\gamma_0 + \gamma_1 z_t)}\ .$$

Things are not so simple now because the parameters γ_0 and γ_1 enter the two terms of L( ) *with the sequence* z_1, ..., z_T. However, the problem can be further simplified by setting $\sigma^2 = \exp(\gamma_0)$; then, the likelihood function becomes

(8) $$L(b, s^2, \gamma | y, X) = \frac{T}{2}\log 2p - \frac{T}{2}\log s^2 - \frac{\gamma}{2}\sum_{t=0}^{T}(z_t) - \frac{1}{2s^2}\sum_{t=1}^{T}\frac{(y_t - x_t'b)^2}{\exp(\gamma z_t)}\ .$$

In this case the number of parameters combined with the individual observations is reduced from two to one, but the problem remains.

Judge, et al. (1988, pp. 374-378) consider an artificially constructed example of a regression with two independent variables and a heteroskedastic disturbance:

$$y = x_1 b_1 + x_2 b_2 + x_3 b_3 + u, \text{ where } x_1 = (1,1,1,\ldots,1)' \text{ and } u_t \overset{iid}{\sim} n(0, s_t^2), \text{ where } s_t^2 = \exp\{g_1 + g_2 x_{2t}\}.$$

There are 20 observations, which have been artificially generated with β = (10, 1, 1)' and α = (-3, 0.03)' . The data are plotted by **basic23.gss**. The results are as follows:

```
-------------------------------------------------------------------------------------------------------------------------
Steepest Ascent Method applied with start value =      0
and maxiter =    100
Gradient at start value(s) =        23.029401
iter =      4
Maximizing value of alpha =       0.21732256
Log Concentrated Likelihood at the maximum =       -61.714772
Gradient Concentrated Likelihood at the maximum =   8.8277323e-005
Hessian Concentrated Likelihood at the maximum =       -111.83176

Runtime for method of steepest ascent =      0   seconds.

Estimates of remaining parameters derived from transformed OLS regression:
beta' =       0.90991102                1.6029533                0.95130962          sigma2 =       0.30277964
Value of the Full Likelihood Function at the Maximum =       -61.714772
Parameter estimates:
     0.90991102               1.6029533               0.95130962               0.30277964               0.21732256
Standard errors implied by the Hessian evaluated at the maximum:
       6.952299                0.387268               0.34183618               0.60429539              0.094572642
Assymptotic t-values implied by the maximum:
     0.13087916               4.1391317                2.7829401               0.50104575                2.2979432
-------------------------------------------------------------------------------------------------------------------------
QNewton applied with start value =              0

=========================================================================
 QNewton Version 3.2.29                          4/19/98  10:05 pm
=========================================================================

return code =    0
normal convergence

Value of objective function       61.714772

Parameters    Estimates    Gradient
-----------------------------------------
alpha           0.2173       0.0000
Number of iterations       4
Minutes to convergence     0.00083
Maximizing value of alpha =       0.21732346
Log Likelihood at the maximum =       -61.714772
Gradient at the maximum =   1.3078068e-005
Hessian at the maximum =       -111.83494
Return code =    0
Runtime for QNewton = 0.05   seconds.
-------------------------------------------------------------------------------------------------------------------------
```

The results from the concentrated likelihood function for the method of steepest ascent and for QNewton, starting from the same value of γ (called alpha in the program and output) are identical.. Both are slightly different than the results obtained by applying the two methods to the full, unconcentrated likelihood function:

```
-------------------------------------------------------------------------------------------------------------------------
Method of steepest ascent applied to the full likelihood:
Steepest Ascent Method applied with start value =    0.9    1.6    0.9    0.3    0.2
and maxiter =    100
Gradient at start value(s) =         1.40848          26.4687                28.4652                17.6061                111.726
iter =      6
Maximizing value of parameters =        0.909876                1.60295                0.95131                0.302774
0.217323
Log Likelihood at the maximum =        -61.7148
```

```
Gradient Likelihood at the maximum =     7.80923e-007  4.43271e-007      -7.4691e-007     0      3.26952e-006
Hessian Likelihood at the maximum =
 -9.502947e-001        -1.742057e+001        -1.848457e+001         0.000000e+000         9.799592e-004
 -1.742057e+001        -3.283667e+002        -3.440132e+002         5.589650e-003         6.980922e-001
 -1.848457e+001        -3.440132e+002        -3.711255e+002         4.036517e-003        -1.830502e+000
  0.000000e+000         5.589650e-003         4.036517e-003        -1.090856e+002        -6.881860e+002
  9.799592e-004         6.980922e-001        -1.830502e+000        -6.881860e+002        -4.454149e+003

Runtime method of steepest ascent =   0.03   seconds.
Maximizing value of parameters =       0.909876         1.60295         0.95131      0.302774            0.217323
Standard errors implied by the Hessian evaluated at the maximum:
      6.90272               0.386378              0.341422               0.60382              0.0945016
Assymptotic t-values implied by the maximum:
      0.131814              4.14867               2.78632               0.501432              2.29968
-------------------------------------------------------------------------------------------------------------------------
QNewton applied to the Full Likelihood Function:
QNewton applied with start value =          0.9                1.6                  0.9                  0.3                  0.2


================================================================================
 QNewton Version 3.2.29                          4/19/98  10:05 pm
================================================================================

return code =    0
normal convergence

Value of objective function      61.714773

Parameters    Estimates    Gradient
-----------------------------------------
beta1          0.9011      -0.0002
beta2          1.6032       0.0000
beta3          0.9515       0.0000
sigma2         0.3028      -0.0000
alpha          0.2173      -0.0002

Number of iterations      16
Minutes to convergence     0.00183
Maximizing value of para =       0.90108939   1.6032202   0.95150162  0.30277953       0.21732251
Log Likelihood at the maximum =      -61.714773
Gradient at the maximum = -1.876719e-004            8.863945e-006         3.733797e-006         -4.693466e-005         -
2.125195e-004
Hessian at the maximum =
-9.503033e-001        -1.742079e+001        -1.848432e+001        -7.102354e-004         0.000000e+000
-1.742079e+001        -3.283668e+002        -3.440133e+002         5.189439e-003         7.658307e-001
-1.848432e+001        -3.440133e+002        -3.711250e+002         4.035635e-003        -1.755174e+000
-7.102354e-004         5.189439e-003         4.035635e-003        -1.090797e+002        -6.881798e+002
0.000000e+000         7.658307e-001        -1.755174e+000        -6.881798e+002        -4.454161e+003
Return code =    0
Runtime for QNewton = 0.11   seconds.
-------------------------------------------------------------------------------------------------------------------------
```

Again, observe that QNewton is much slower than the method of steepest ascent even when the same start values are used.

The program and full output are given in an appendix to these notes. Included are graphs of the full likelihood function in the vicinity of the maximum sliced in the direction of one parameter at a time. These and the above results give no clue that anything is amiss, but a 3-D graph of γ versus σ^2 (alpha versus sigma2 in the program) show that the likelihood surface is truely badly behaved. Two-dimensional appearances may be deceptive. Nonetheless, neither numerical maximization of the concentrated or the full likelihood fuction ran into difficulty.

(4) *Regression with Box-Cox Transformations.* This transformation was introduced in 1964 by G.E.P. Box and D.R. Cox, "An Analysis of Transformations," *Jour. Royal Statistical Society, Ser. B, 26*: 211-243 (1964).

For a strictly positive variable x, the transform is

$$(9) \qquad x^{\lambda} = (x^{\lambda} - 1)/\lambda, \quad \lambda \neq 0$$
$$= \log x, \quad \lambda = 0.$$

It includes linearity, $\lambda = 1$, and log-linearity, $\lambda = 0$, as special cases. The case $\lambda = -1$ is the reciprocal transformation. In its most general form, a Box-Cox transformed regression is

$$(10) \qquad y_t^{(\lambda_\theta)} = \alpha + \beta_1 x_{1t}^{(\lambda_1)} + \beta_2 x_{2t}^{(\lambda_2)} + \ldots + \beta_k x_{k_t}^{(\lambda_k)} + \varepsilon_t .$$

But a form so general is rarely estimated. Rather λ is generally restricted to be either 1 or the same for every variable to which it is applied. The three leading cases are thus:

Case 1: $y = \alpha + \beta_1 x_{1t}^{(\lambda)} + \beta_2 x_{2t}^{(\lambda)} + \ldots + \beta_k x_{kt}^{(\lambda)} + \varepsilon_t .$

Case 2: $y^{(\lambda)} = \alpha + \beta_1 x_{1t} + \beta_2 x_{2t} + \ldots + \beta_k x_{kt} + \varepsilon_t .$

Case 3: $y^{(\lambda)} = \alpha + \beta\, x_{1t}^{(\lambda)} + \beta_2 x_{2t}^{(\lambda)} + \ldots + \beta_k x_{kt}^{(\lambda)} + \varepsilon_t .$

In many cases, however, we would want to transform the dependent variable differently from at least some of the independent variables. This can be done relatively easily provided there are only *two* different values for λ in addition to 1.0. There is a fundamental difference, as we shall see, between case 1, in which only the independent variables are transformed, and the others. It suffices to treat the following case to show the difference and to discuss estimation and inference issues:

$$(11) \qquad y_t^{(\theta)} = \alpha + \beta x_t^{(\lambda)} + \varepsilon_t ,$$

where $\theta \neq \lambda$, $y_t, x_t > 0$ all t and where $\varepsilon_t \overset{i.i.d.}{\sim} n(0, \sigma^2)$. Note the inconsistency between the assumption $y_t > 0$ all t and $\varepsilon_t \sim n(0, \sigma^2)$; this can cause convergence problems for several estimation methods.

Under the assumption that $\varepsilon_t \overset{i.i.d.}{\sim} n(0, \sigma^2)$ for the true value of λ and θ, the density for a sample of size T, $(\varepsilon_1, \ldots, \varepsilon_T)'$, is

$$(12) \qquad \varepsilon \sim \frac{1}{(2\pi\sigma^2)^{T/2}} \exp\left\{\frac{-1}{2\sigma^2} \sum_{t=1}^{T} \varepsilon_t^2\right\}.$$

Now the Jacobian of the transform from ε_t to y_t, given x_t is

$$(13) \qquad J(\varepsilon_t \rightarrow y_t) = |y^{\theta-1}| = y^{\theta-1},$$

if $y > 0$. Thus the joint density of $y = (y_1, \ldots, y_T)'$ conditional on $x = (x_1, \ldots, x_T)'$ is

$$(14) \qquad f(y|x) = \left(\frac{1}{2\pi\sigma^2}\right)^{T/2} \exp\left\{\frac{-1}{2\sigma^2} \sum_{t=1}^{T} \left(y_t^{(\theta)} - \alpha - \beta x_t^{(\lambda)}\right)^2\right\} \prod_{t=1}^{T} y_t^{\theta-1} .$$

It follows that the log likelihood function is

$$(15) \qquad L(\alpha, \beta, \sigma^2, \theta, \lambda|y, x) = k - \frac{T}{2} \log \sigma^2 + (\theta - 1) \sum_{t=1}^{T} \log y_t - \frac{1}{2\sigma^2} \sum_{t=1}^{T} \left(y_t^{(\theta)} - \alpha - \beta x_t^{(\lambda)}\right)^2 .$$

Concentrate this likelihood function with respect to σ^2:

$$(16) \qquad \hat{\sigma}^2 = S(\alpha, \beta, \theta, \lambda|y, x) = \frac{1}{T} \sum_{t=1}^{T} (y^{(\theta)} - \alpha - \beta x_t^{(\lambda)})^2 ,$$

hence

$$(17) \qquad L^*(\alpha, \beta, \lambda, \theta|y, x) = k^* - \frac{T}{2} \log S(\alpha, \beta, \lambda, \theta) + (\theta - 1) T \overline{\log y} ,$$

where $\overline{\log y} = \frac{1}{T} \sum_{t=1}^{T} \log y_t$ is the geometric mean of the dependent variable.

If $\theta = 1$, ML estimation of α, β, and λ is equivalent to minimizing the sum of squares

$$S(\alpha, \beta, \lambda|y, x) = \frac{1}{T} \sum_{t=1}^{T} \left(y_t - \alpha - \beta x_t^{(\lambda)}\right)^2 .$$

For a fixed value of λ this is equivalent to estimating α and β from the OLS regression of y_t on $x_t^{(\lambda)}$. Hence to find $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\lambda}$, we can iterate or do a grid search on λ using OLS to concentrate S with respect to α and β.

But for $q \neq 1$, the likelihood function involves another term,

$$(\theta - 1) T \overline{\ln y} ,$$

so that ML is *not* equivalent to

$$\min_{\alpha,\beta,\theta,\lambda} S(\alpha, \beta, \theta, \lambda | y, x).$$

Moreover, $y_t^{(\theta)}$ may not exist for some possible values of ε_t, depending on the value of θ.

If $\lambda = \theta$ is assumed, a grid-search on λ, usually in the interval [-2, 2] is easy to implement. The criterion function in this case is

$$(\lambda - 1)\, T\, \overline{\log y} - \frac{T}{2} \log S(\alpha, \beta, \lambda) = C(\lambda). \tag{18}$$

For given λ, regress

$$\frac{y_t^{\lambda} - 1}{\lambda} \text{ on } \frac{x_t^{\lambda} - 1}{\lambda}$$

to obtain $\hat{\alpha}_{(\lambda)}$ and $\hat{\beta}(\lambda)$. Calculate

$$C(\lambda) = (\lambda - 1) T\, \overline{\log y} - \frac{T}{2} \log S(\hat{\alpha}(\lambda), \hat{\beta}(\lambda), \lambda).$$

Change λ in such a way as to locate the maximum C.

An asymptotic variance-covariance matrix for these estimates may be obtained from the likelihood function in the usual way for the ML estimates:

Let

$$\varepsilon_t = y_t^{(\lambda)} - \beta x_t^{(\lambda)} - \alpha. \tag{19}$$

Then

$$\frac{\partial f(y_t)}{\partial \beta} = \frac{\varepsilon_t}{\sigma^2} x_t^{(\lambda)}, \qquad \frac{\partial f(y_t)}{\partial \alpha} = \frac{\varepsilon_t}{\sigma^2} \cdot 1$$

$$\frac{\partial f(y_t)}{\partial \lambda} = \log y_t - \frac{\varepsilon_t}{\sigma^2} \left(\frac{\partial y^{(\lambda)}}{\partial \lambda} - \beta \frac{\partial x_t^{(\lambda)}}{\partial \lambda} \right) \tag{20}$$

$$\frac{\partial f(y_t)}{\partial \sigma^2} = \frac{1}{2\sigma^2} \left(\frac{\varepsilon_t^2}{\sigma^2} - 1 \right).$$

Moreover, the derivatives $\frac{\partial y_t^{(\lambda)}}{\partial \lambda}$ and $\frac{\partial x_t^{(\lambda)}}{\partial \lambda}$ are given by

$$\frac{\partial y_t^{(\lambda)}}{\partial \lambda} = \frac{1}{\lambda} \left(y_t^{\lambda} \log y_t - \left[\frac{y_t^{\lambda} - 1}{\lambda} \right] \right)$$

$$\frac{\partial x_t^{(\lambda)}}{\partial \lambda} = \frac{1}{\lambda} \left(x_t^{\lambda} \log x_t - \left[\frac{x_t^{\lambda} - 1}{\lambda} \right] \right). \tag{21}$$

The estimated asymptotic variance-covariance matrix, V - 1, is obtained from (22) by inserting $\hat{\alpha}, \hat{\beta}, \hat{\lambda}, \text{and } \hat{\sigma}^2$ in (23) and forming the matrix

$$V = \begin{pmatrix} \frac{\partial f}{\partial \alpha} \\ \frac{\partial f}{\partial \beta} \\ \frac{\partial f}{\partial \lambda} \\ \frac{\partial f}{\partial \sigma^2} \end{pmatrix} \left(\frac{\partial f}{\partial \alpha} \quad \frac{\partial f}{\partial \beta} \quad \frac{\partial f}{\partial \lambda} \quad \frac{\partial f}{\partial \sigma^2} \right).$$

Obviously, this can be done numerically as well as analytically. In "Optimization in GAUSS: A Tutorial," I will explore and compare analytical and numerical hessians and gradients.

In this example, I consider finding the maximum of the 2-parameter likelihood function for the example of estimation of a relationship involving the Box-Cox transformation given in Judge, *et al., op. cit.*, pp. 555-563. The data are from Table 12.7, p.561. The data, 40 observations, were generated from the following model:

$$y^{(q)} = b_0 + b_1 x_1^{(l_1)} + b_2 x_2^{(l_2)} + e,$$

$$\text{where } e \underset{\sim}{} iid\ n(0,2.25) \text{ and } y^{(q)} = \frac{y^q - 1}{q},\ x_i^{(l_i)} = \frac{x_i^{l_i} - 1}{l_i},\ \text{i} = 1,2.$$

They do not say how the x's were generated. In my example, I set $\lambda_1 = \lambda_2 = \lambda$ and estimate only a single right-hand transformation parameter. Note that the constant term is not transformed.

basic24.gss, included in the appendix estimates and graphs the results for this problem using the method of steepest ascent and QNewton on both the concentrated and the unconcentrated likelihood functions. QNewton does not converge when applied to the full, unconcentrated likelihood function even when started at the known maximizing values. Although the concentrated likelihood function (17) is well-behaved in θ and λ, at the maximizing values of *all* of the parameters the unconcentrated likelihood is very poorly behaved as the graphs generated by **basic24.gss** demonstrate. There is thus an important difference between *concentrating* the likelihood function and *slicing* the likelihood function, which I wlll explore in "Optimization in GAUSS: A Tutorial." I would venture the guess that the convergence difficulties of QNewton in the case of the unconcentrated likelihood function are due to this problem.

For the concentrated likelihood function the results generated by **basic24.gss** are as follows:

```
Steepest Ascent Method applied with start value =      0      0
and maxiter =    100
Gradient at start value(s) =         93.91556               0
iter =     27
Maximizing value of box-cox parameters =        1.0513175            -0.93256424
Log Concentrated Likelihood at the maximum =       -79.430749
Gradient Concentrated Likelihood at the maximum =    0.00021221983      -0.00074668516
Hessian Concentrated Likelihood at the maximum =
     -2.3895977             0.059293259
     0.059293259            -0.5387602

Runtime for method of steepest ascent =   0.13   seconds.

Estimates of remaining parameters derived from transformed OLS regression:
beta' =      -8.4195207             14.66048             9.9569271          sigma2 =       3.9536372
Value of the Full Likelihood Function at the Maximum =       -79.430749
Parameter estimates:
      1.0513175           -0.93256424           -8.4195207            14.66048            9.9569271            3.9536372
Standard errors implied by the Hessian evaluated at the maximum:
     0.71818888            1.4255939             28.598397            38.362667            25.241539            13.370241
Asymptotic t-values implied:
      1.4638454           -0.65415841           -0.29440533           0.38215486           0.39446592           0.29570427
-------------------------------------------------------------------------------------------------------------------------------
Standard errors from the variance-covariance matrix using the outer product method.
Reference: Davidson & MacKinnon, pp.265-267.
Sum of gradients evaluated at individual observations:
   0.00019276775       -0.00074058978        4.3778507e-007       -4.8769456e-007       -3.1889737e-007       -4.4929688e-008

Condition number of the hessian evaluated by the outer product method =        25943410
Parameter estimates:
      1.0513175           -0.93256424           -8.4195207            14.66048            9.9569271            3.9536372
Standard errors implied by the Hessian evaluated by the outer product method:
      1.1824471            1.4470135             40.577471            52.721719            35.768356             21.51694
Asymptotic t-values implied:
      0.8891032           -0.64447513           -0.20749249           0.27807287           0.27837251           0.18374533
-------------------------------------------------------------------------------------------------------------------------------
QNewton applied with start value =                1                   -1
Note: QNewton converges to the wrong values for start = {0.0, 0.0}. Try it!

================================================================================
```

```
 QNewton Version 3.2.29                    4/21/98   9:56 am
================================================================================

return code =    0
normal convergence

Value of objective function      79.430748

Parameters    Estimates    Gradient
-------------------------------------------
theta          1.0514       0.0000
lambda         -0.9337      0.0001

Number of iterations       8
Minutes to convergence     0.00150
Maximizing value of para2 =       1.0513881              -0.93374683
Log Likelihood at the maximum =       -79.430748
Gradient at the maximum =   2.8384185e-005            0.00010653421
Hessian at the maximum =
     -2.3903285             0.059608948
    0.059608948             -0.53828537
Return code =    0
Runtime for QNewton = 0.11   seconds.
--------------------------------------------------------------------------------------------------------------------------------
```

While for the full, unconcentrated likelihood function, **basic24.gss** yields:

```
Method of steepest ascent applied to the full likelihood:
Steepest Ascent Method applied with start value =   1.05   -0.9   -8.5     15     10      4
and maxiter =    100
Gradient at start value(s) =         107.407         -66.547          -5.91729          -4.74346          -4.61087
0.358554
iter =     10
Maximizing value of parameters =         1.05136      -0.933938       -8.4385        14.6874        9.97424        3.95433
Log Likelihood at the maximum =         -79.4307
Gradient Likelihood at the maximum =   -9.191358e-004          4.823491e-004         4.429053e-005         3.454166e-005
3.348175e-005           7.187492e-006
Hessian Likelihood at the maximum =
 -3.630436e+003         1.913560e+003         1.766268e+002         1.390596e+002         1.357543e+002         2.375314e+001
  1.913560e+003        -1.149939e+003        -1.064531e+002        -8.369105e+001        -8.168502e+001        -7.345713e-004
  1.766268e+002        -1.064531e+002        -1.011546e+001        -7.850663e+000        -7.690003e+000        -9.291349e-005
  1.390596e+002        -8.369105e+001        -7.850663e+000        -6.156883e+000        -5.957937e+000         6.005527e-005
  1.357543e+002        -8.168502e+001        -7.690003e+000        -5.957937e+000        -5.925299e+000         1.965187e-005
  2.375314e+001        -7.345713e-004        -9.291349e-005         6.005527e-005         1.965187e-005        -1.278982e+000

Runtime method of steepest ascent =    0.2   seconds.
Maximizing value of parameters =         1.05136      -0.933938       -8.4385        14.6874        9.97424        3.95433
Standard errors implied by the Hessian evaluated at the maximum:
       0.690377              1.39635              26.9402              36.1817              23.7894              12.8549
Assymptotic t-values implied by the maximum:
        1.52287             -0.668843             -0.313231              0.405935              0.419273              0.307613
--------------------------------------------------------------------------------------------------------------------------------
QNewton applied to the Full Likelihood Function does not converge
            even when started at the known maximizing values!
QNewton applied with start value =         1.05132          -0.932564             -8.41952                14.6605
9.95693          3.95364
Gradient computed from GRADP at start values =
2.514197e-004         -7.741144e-004         -2.869338e-006         -3.004925e-006         -2.854466e-006         3.594375e-007
Value of the joint likelihood at the start values =       -79.430749
--------------------------------------------------------------------------------------------------------------------------------
```

(5) *Nonlinear Regression.* In this example, I consider the maximum-likelihood (identical to nonlinear least-squares) estimates of the coefficients for the simplest form of nonlinear regression:

$$y = a_0 + a_1 x^b + e, \quad e \text{ iid } n(0, s^2) .$$

In this case, I generated the data: N = 20 observations, with $a_0 = 2.5$, $a_1 = 0.25$, b = 2.5, $\sigma^2 = 100$. The x series was also generated by **x = 0.5*seqa(1, 1,20) + sqrt(10)*rndu(N,1);** . The resulting series was then conditioned on in forming the likelihood function:

{3.5104 1.7278 3.1601 4.0099 3.4259 4.1231 4.7934 4.7501 5.0277 7.2197
7.5301 7.6453 7.4881 7.7315 9.6487 10.2315 10.8605 11.0478 11.2747 10.1609}' .

The results of generating a sample of y's for these parameters and values of x are generated and graphed by **basic25.gss**. It is apparent from Figure 1, presented as part of the output of **basic25.gss** in the appendix, that the relationship to be estimated is quite nonlinear. The likelihood function can easily be concentrated in the parameter b so that the problem can be reduced to maximizing the concentrated likelihood function with respect to the single parameter b and then recovering estimates of a_0, a_1, and σ^2 by regressing y on a vector of ones and $x^{\hat{b}}$, where $\hat{b}$ is that value which maximizes the concentrated likelihood function. I used the method of steepest ascent. The concentrated likelihood function is graphed in Figure 2; it has a beautiful shape, but, as remarked above, 2-D appearances are deceptive. In fact, when a_1 and σ^2 are also allowed to vary, the interaction can cause convergence problems for both QNewto and the method of steepest ascent applied to the full, unconcentrated likelihood function. The results for the concentrated likelihood functionm are as follows:

```
-------------------------------------------------------------------------------------------------------------------------------
Maximize the Concentrated Likelihood Function using Method of Steepest Ascent:
Start =       2
Steepest Ascent Method applied with start value =      2
and maxiter =    100
Gradient at start values =
  4.45937
iter =     5
Maximizing value of beta =          2.8261
Concentrated Likelihood at the maximum =         -73.1343
Hessian at the maximum =         -4.9347
Remaning parameters evaluated by LS(beta)
Parameters =         6.81142             0.120972                  2.8261                  92.4663
Gradient at the maximum =
           0   -2.34944e-005          -5.17929e-005           -0.00540736
Asymptotic Standard errors implied by the Hessian evaluated at the maximum:
      5.14802                 0.137378                0.464789                 30.8229
Assymptotic t-values implied by the maximum:
      1.32312                 0.880576                6.08039                  2.99992
Runtime for method of steepest ascent =     0.01   seconds
-------------------------------------------------------------------------------------------------------------------------------
Maximize the Concentrated Likelihood Function using QNewton:

================================================================================
 QNewton Version 3.2.29                          4/21/98  11:23 pm
================================================================================

return code =    0
normal convergence

Value of objective function      73.134285

Parameters    Estimates    Gradient
-----------------------------------------
betahat         2.8261       0.0000
Number of iterations       5
Minutes to convergence     0.00050
betahat = 2.8260973
Value of the concentrated likeliood function = -73.134285
Gradient = 2.5142189e-006
Hessian = -4.9346015
Runtime for QNewton =     0.05   seconds
-------------------------------------------------------------------------------------------------------------------------------
```

Both methods yield the same estimate of b, 2.862, from the same start value 2, but steepest ascent converges in one-fifth of the time that QNewton takes.

The remaining parameters may be recoverd by OLS and the appropriate asymptotic standard errors and t-statistics can then be obtained from the hessian of the full likelihood fuction evaluated at the estimates While the estimates of the parameters are not really close to the true values used to generate the data, 20 observations is not a great many for the good asymptotic properties of ML to be expected to "kick in." The fit is not particularly good and the residuals show considerable evidence of hetereoskedasticity. On the other hand b is reasonably well and accurately estimated and the asymptotic t-statistics for the other

parameters clearly indicate the unreliable nature of the results. Graphs of the nonlinear regression equation and the actual values of y and of the residuals from the regression, not presented here, both suggest that the regression captures the nonlinearity well (est b is highly significant and not far from the true value), but yield strong evidence of heteroskedasticity not present in the DGP and not correctable by the standard fix-up.

Alternatively, one may apply both methods to the full, unconcentrated likelihood function involving four parameters. It turns out to be surprising difficult to achieve convergence for either method. Some of the reasons for the problem are suggested by the graphs of the likelihood function "sliced" in the direction of b and a_1 and of b and σ^2 in the vicinity of the maximum; the one-directional slices appear to show a well-behaved likelihood function, however. The numerical results are as follows:

```
---------------------------------------------------------------------------------------------------------------------
Steepest Ascent Method applied with start values =    6.8          0.12         2.8          92
and maxiter =    100
Note that convergence of steepest ascent is very sensitive
            to the start values in this case.
Gradient at start values =
  0.562071    349.491     97.8002   0.0113892
iter =     7
Maximizing value of parameters =
       6.81134              0.120975              2.82609               87.843
Compare these with values obtained from the concentrated likelihood function:
            6.81142              0.120972              2.8261               92.4663.
 logLikelihood at the maximum =        -73.1343
Gradient at the maximum =
  -2.08635e-007                 0     3.01707e-006                 0
Standard errors implied by the Hessian evaluated at the maximum:
       5.0174               0.133906              0.453034              27.7793
Assymptotic t-values implied by the maximum:
       1.35754              0.903428              6.23814               3.16218
Runtime method of steepest ascent  0.02   seconds.
---------------------------------------------------------------------------------------------------------------------
Maximize the Full Likelihood function using QNewton:
with start values:       6            0.1          2           90

=============================================================================
 QNewton Version 3.2.29                    4/21/98  11:23 pm
=============================================================================

return code =    0
normal convergence

Value of objective function      73.134285

Parameters    Estimates    Gradient
-------------------------------------------
a0             6.8114      -0.0000
a1             0.1210      -0.0007
b              2.8261      -0.0002
sigma2         87.8433      0.0000

Number of iterations      62
Minutes to convergence     0.00250
Value of the  likeliood function =     -7.3134e+001
Gradient =     -4.1727e-007          -6.6958e-004          -1.7901e-004           3.3973e-007
return code =    0
Runtime for Qnewton applied to the unconcentrated likelihood function =     0.15   seconds
---------------------------------------------------------------------------------------------------------------------
Parameter values from QNewton:
       6.81137              0.120974              2.82609               87.8433
Graphs of the Joint Likelihood Function in the Vicinity of the Maximum:
One at a time, holding other parameters at their maximizing values,
then the pair sigma2 vs. alpha, holding the regression parameters at their
maximizing values.
Values of parameters at the maximum:
       6.81137              0.120974              2.82609               87.8433
---------------------------------------------------------------------------------------------------------------------
```

QNewton is much less sensitive to the start values. The much greater length of time it takes to run, however, is partly due to the fact that I started farther away from the maximizing values.

b. An Example of Quadratic Programming using Qprog

A standard quadratic programming problem is the following:

minimize $0.5 * x'Qx - x'R$
subject to constraints,
$Ax = B$
$Cx \geq D$
and bounds on the x's,
$bnds[.,1] \leq x \leq bnds[.,2]$

For example, least squares regression with equality and inequality constraints and bounded x's is a case of quadratic programming.

QProg solves the quadratic programming problem specified above.

Format:
{ x,u1,u2,u3,u4,ret } = QProg(start,q,r,a,b,c,d,bnds);

Input:	start	K×1 vector of starting values.
	q	K×K matrix, model coefficient matrix
	r	K×1 vector, model constant vector
	a	M×K matrix, equality constraint coefficient matrix if no equality constraints in model, set to zero
	b	M×1 vector, equality constraint constant vector, if set to zero and M > 1, b is set to M×1 vector of zeros
	c	N×K matrix, inequality constraint coefficient matrix, if no inequality constraints in model, set to zero
	d	N×1 vector, inequality constraint constant vector if set to zero and N > 1, d is set to Mx1 vector of zeros
	bnds	K×2 vector, bounds on x, the first column contains the lower bounds on x, and the second column the upper bounds, if zero, bounds for all elements of x default to ±1e200
Output:	x	K×1 vector, coefficients at the minimum of the function
	u1	M×1 vector, Lagrangian coefficients of equality constraints
	u2	N×1 vector, Lagrangian coefficients of inequality constraints
	u3	K×1 vector, Lagrangian coefficients of lower bounds
	u4	K×1 vector, Lagrangian coefficients of upper bounds
	ret	return code: 0, successful termination 1, max iterations exceeded 2, machine accuracy is insufficient to maintain decreasing function values 3, model matrices not conformable <0, active constraints inconsistent
Global:	**_qprog_maxit**	scalar, maximum number of iterations, default = 1000

The folowing example appears in a 1960 paper by Houthakker: [35]

maximize $18x_1+16x_2+22x_3+20x_4-3x_1^2-x_1x_2-8x_1x_3-5x_2^2-x_2x_3-4x_2x_4-8.5x_3^2-3x_3x_4-5.5x_4^2$
subject to $-x_1\leq 0$, $-x_2\leq 0$, $-x_3\leq 0$, $-x_4\leq 0$, $5x_1+10x_3\leq 2$, $4x_2+5x_4\leq 3$, $x_1+x_2+x_3+x_4\leq 5/3$.

[35] Houthakker, H. S., "The Capacity Method of Quadratic Programming," *Econometrica, 28*: 62 - 87 (1960).

It is solved in **basic26.gss**. Here is my solution compared with Houthakker's:

```
---------------------------------------------------------------------------------------------------------------------------------------------------------
x' =          0.4                  0.233083              3.15986e-033          0.413534
Houthakker's solution:
              0.4                  0.233083              0                     0.413534
Lagrangian coefficients for the constraints:
u1' =         0
u2' =         0              0        13.4075        0        3.07338     2.90376        0
u3' =         0              0           0           0
u4' =         0              0           0           0
ret =         0
Maxval =    -14.858
---------------------------------------------------------------------------------------------------------------------------------------------------------
```

My solution is essentially the same as Houthakker's although he did not obtain the Lagrangian coefficients.

This concludes the tutorial "Notes on GAUSS." A more advanced tutorial "Optimization in GAUSS: A Tutorial" will follow at a later date.

APPENDIX:

PROGRAMS BASIC01 - BASIC26

AND ASSOCIATED OUTPUT