A GAUSS primer

Curt Wells

Lund University
Fall Term, 1998

1 Introduction

This is a short paper on the subject of GAUSS, not the German mathemetician but rather the computer program. This program is really a high–level, matrix oriented language. It differs from programs such as TSP or RATS in that one cannot push a button and get an answer: one has to tell GAUSS what to do. This is not as formidible as one might suppose as may people work with the program and thus there is lots of code floating around that can be adapted to your specific problem.

Basically, the choice between GAUSS and some other econometrics package will hinge on the nature of the problem you confront. With standard problems the packaged solution should be prefered; with non–standard problems, these packages are often difficult to program and slow. For example, if you want to estimate a Poisson count model, LIMDEP is for you. Even if you have non–linearities, calculating interactions and their variance is straight forward using the matrix commands in the program. However, if you wish to esitmate a Poisson Hurdle model, then you must program it yourself in LIMDEP and the advantage that this program has over GAUSS is lost. Indeed, doing a hurdle model is much more difficult in LIMDEP.

This paper will not teach you how to program. Writing good code is an art that I cannot teach; one might also question the "artiness" of my code. However, it (usually) works and that's all I care about.

This paper will present the basic facts about GAUSS . Section 2 will take up the basic Gauss language; Section 4 describes functions and proceedures; Section 5 presents the concept of Libraries; Section 6 deals with input and output from GAUSS ; and Section 7 considers graphics. Finally, Section 8 is devoted to an example using maximum likelihood on a problem that one might well meet in research.

One final comment before we start. You will find GAUSS on *Tellus* at \Win\prg\stat\gauss\gauss.32. You start GAUSS by clicking on the icon — or on *gauss.exe*. Commands my be entered directly on the Gauss window [1] Typing a carriage–return, ↵, will execute the statement. Note that at the moment (GAUSS 3.2.32) only last typed statement can be executed.[2] Once you have written a few lines, you may wish to save them in a file for later use. The statements may then be copied and pasted into a file of ones choice. For instance, after hitting the *copy* button, enter the name of a destination file on the space beside the *edit* button and push the *edit* button. Now you must remove the (gauss) prompt from the file and add a semicolon (;) after each statement. Once this is done, you may hit the ">" beside the *run* button (the name of the file will now appear beside the button), and press the *run* button. This will execute the commands.

I use the command window as a pocket calculator and to check on the correctness of some statements. Otherwise I always write a small file with

[1] This is usually called the *command window*.

[2] This means that you cannot move the cursor to a previously typed row and push "↵" and get an executed statement. Instead you must cut and paste this statement to the last row: the one with the cursor agfer the prompt(gauss). This is a pain but there is nothing we can do about it.

the statements and submit them from the editor. [3] One last remark before beginning: here I use printers' quotation marks in GAUSS statements contained in the text. This is to make my document look nice. In programs, you of course must use the straigt quotes ' " ' when typing your programs.

2 The GAUSS Language

As GAUSS is a mathematical language, one must know and be able to use the operators. The following pages list and exemplify all of the operators. The section concludes with a table listing the operators more compactly. Note that most of the text has been scanned from the GAUSS manual.

2.1 Element–by–Element Operators

The operators described as Element–by-Element operators share common rules of conformability. Some functions that have two arguments also operate according to the same rules.

The Element–by–Element operators handle those situations in which matrices are not conformable according to standard rules of matrix algebra. When a matrix is said to be ExE conformable, it refers to this Element–by–Element conformability. The following cases are supported:

matrix	*operation*	matrix
matrix	*operation*	scalar
scalar	*operation*	matrix
matrix	*operation*	vector
vector	*operation*	matrix
vector	*operation*	vector

In a typical expression involving an Element–by-Element operator,

$$z = X + Y;$$

conformability is defined as follows:

- If z and y are the same size, the operations are carried out corresponding element by corresponding element.

$$x = \begin{matrix} 1 & 3 & 2 \\ 4 & 5 & 1 \\ 3 & 7 & 4 \end{matrix}$$

$$y = \begin{matrix} 2 & 4 & 3 \\ 3 & 1 & 4 \\ 6 & 1 & 2 \end{matrix}$$

[3] Note that you cannot highlight a few statements and run these as one can in SAS or RATS or LIMDEP. As the GAUSS editor is rather limited — goodies such as 'undo', while on the menu, do not work — I use a text editor called programmer's file editor which is available free from *http://www.lancs.ac.uk/people/cpaap/pfe/*.

$$x + y = \begin{matrix} 3 & 7 & 5 \\ 7 & 6 & 5 \\ 9 & 8 & 6 \end{matrix}$$

- If x is a matrix and y is a scalar, or vice versa, then the scalar is operated on with respect to every element in the matrix. For example, $x + 2$ will add 2 to every element of x.

$$x = \begin{matrix} 1 & 3 & 2 \\ 4 & 5 & 1 \\ 3 & 7 & 4 \end{matrix}$$

$$y = 2$$

$$x + y = \begin{matrix} 3 & 5 & 4 \\ 6 & 7 & 3 \\ 4 & 9 & 6 \end{matrix}$$

- If x is an N by 1 column vector and y is an N by K matrix, or vice versa, the vector is swept 'across" the matrix.

vector		**matrix**		
1	$\longrightarrow$	2	4	3
4	$\longrightarrow$	3	1	4
3	$\longrightarrow$	6	1	2
				result
		3	5	4
		7	5	8
		9	4	5

- If x is an l by K column vector and y is an N by K matrix, or vice versa, then the vector is swept "down" the matrix.

vector	2	4	3
	$\downarrow$	$\downarrow$	$\downarrow$
	1	1	1
matrix	3	1	4
	6	1	2
	4	8	6
result	5	5	7
	8	5	5

- When one argument is a row vector and the other is a column vector, the result of an Element–by–Element operation will be the **table** of the two.

row vector		2	4	3	1
	3	5	7	6	4
column vector	2	4	6	5	3
	5	7	9	8	6

- If z and y are such that none of these conditions apply, then the matrices are not conformable to these operations and an error message will be generated.

2.2 Matrix Operators

The following operators work on matrices. Some assume numeric data and others will work on either character or numeric data.

2.2.1 Numeric Operators

See Section 2.1 for details on how matrix conformability is defined for Element–by–Element operators.

\+ Addition:y $y = x + z$; Performs Element–by–Element addition.

\- Subtraction or negation:

$$y = x - z; y = -k;$$

Performs Element–by-Element subtraction or the negation of all elements, depending on context.

* Matrix multiplication or multiplication: $y = x * z$; When z has the same number of rows as x has columns, this will perform matrix multiplication (inner product). If x or z are scalar, this performs standard Element–by–Element multiplication.

/ Division or linear equation solution: $z = b/A$; If A and b are scalars, it performs standard division. If one of the operands is a matrix and the other is scalar, the result is a matrix the same size with the results of the divisions between the scalar and the corresponding elements of the matrix. Use ./ for Element–by–Element division of matrices.
If b and A are conformable, this operator solves the linear matrix equations:

$$Ax = b;$$

Linear equation solution is performed in the following cases:
1. If A is a square matrix and has the same number of rows as b, then this statement will solve the system of linear equations using an LU decomposition.
2. If A is rectangular with the same number of rows as b, then this statement will produce the least squares solutions by forming the normal equations and using the Cholesky decomposition to get the solution.

$$y = \frac{A'b}{A'A}$$

/ *(continued)* If TRAP 2 is set, missing values will be handled with pairwise deletion. For least squares solutions, all computations and storage of intermediate results are done in 80-bit precision. PRCSN 64 can be specified to force 64-bit precision for smaller matrices or when using the 32-bit version of GAUSS.

% Modulo division: $y = x\%z$; For integers, this returns the integer value that is the remainder of the integer division of z by x. If x or z are noninteger, they will first be rounded to the nearest integer. This is an element-by-element operator.

! Factorial: $y = x!$; Computes the factorial of every element in the matrix r. Nonintegers are rounded to the nearest integer before the factorial operator is applied. GAUSS-386i: This will not work with complex matrices. If x is complex, a fatal error will be generated.

.* Element-by-element multiplication: $y = x. * z$; If x is a column vector, and z is a row vector (or vice versa), then the "outer product" or "table" of the two will be computed. See Section 2.1 for conformability rules.

./ Element–by–element division: $y = x./z$;

^ Element–by–element exponentiation: $y = x\hat{}z$; If x is negative, z must be an integer.

.^ Same as ^.

.*. Kronecker(tensor) product: $y = x. * .z$; This results in a matrix in which every element in x has been multiplied (scalar multiplication) by the matrix z. For example:

```
x = {1 2, 3 4};
z = {4 5 6, 7 8 9};
y = x .*. z;
```

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

$$z = \begin{matrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

$$y = \begin{matrix} 4 & 5 & 6 & 8 & 10 & 12 \\ 7 & 8 & 9 & 14 & 16 & 18 \\ 12 & 15 & 18 & 16 & 20 & 24 \\ 21 & 24 & 27 & 28 & 32 & 36 \end{matrix}$$

*~ Horizontal direct product: $z = x*\tilde{}y$;

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

$$y = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$$

$$z = \begin{matrix} 5 & 6 & 10 & 12 \\ 21 & 24 & 28 & 32 \end{matrix}$$

The input matrices x and y must have the same number of rows. The result will have cols(x) * cols(y) columns.

2.2.2 Other Matrix Operators

There are a number of other matrix operators that are very useful. These are listed below

’ Transpose operator. In the expression $y = x$’; the columns of y will contain the same values as the rows of x and the rows of y will contain the same values as the columns of x. **GAUSS–386i**: For complex matrices this computes the complex conjugate transpose. If an operand immediately follows the transpose operator, the ’ will be interpreted as ’*. Thus y = x’x is equivalent to y = x’*x.

.’ Bookkeeping transpose operator: $y = x$.’; This is provided primarily as a matrix handling tool for **GAUSS–386i**. For all matrices, the columns of y will contain the same values as the rows of x and the rows of y will contain the same values as the columns of x. The complex conjugate transpose is NOT computed for complex matrices when you use .’.
If an operand immediately follows the bookkeeping transpose operator, the .’ will be interpreted as .’*. Thus $y = x$.’x is equivalent to $y = x$.’*x.

| Vertical concatenation

$$z = x|y;$$

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix}$$

$$y = \begin{matrix} 7 & 8 & 9 \end{matrix}$$

$$z = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

~ Horizontal concatenation

$$z = x\tilde{}y;$$

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

$$y = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$$

$$z = \begin{matrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{matrix}$$

2.3 Relational Operators

See Section 2.1 for details on how matrix conformability is defined for Element–by–Element operators.

Each of these operators has two equivalent representations. Either can be used (eg., < or LT), depending only upon preference. The alphabetic form should be surrounded by spaces.

A third form of these operators has a '$' and is used for comparisons between character data and for comparisons between strings.

In addition, these '$' comparison operators can be used to do comparisons on missing values or infinities or any other type of NaN's that are possible in the IEEE double precision format. Less than or greater than comparisons may not be very meaningful with NaN's, but equal and not equal will be valid. The comparisons are done byte by byte starting with the lowest addressed byte of the elements being compared.

If the relational operator *is not* preceded by a dot '.', then the result is always a scalar 1 or 0, based upon a comparison of all elements of x and y. All comparisons must be true for the relational operator to return TRUE.

By this definition, then

$$\textbf{if } x/=y;$$

is interpreted as: "is every element of z not equal to the corresponding element of y". To check if two matrices are not identical, use:

$$\textbf{if not } x==y;$$

GAUSS–386i: For complex matrices, the '==', '/=', '.==' and './=' operators compare both the real and imaginary parts of the matrices; all other relational operators compare only the real parts.

- Less than

$$\begin{array}{cccccc} z & = & x & < & y; \\ z & = & x & \textbf{LT} & y; \\ z & = & x & \$< & y; \end{array}$$

- Less than or equal to

$$\begin{array}{cccccc} z & = & x & <= & y; \\ z & = & x & \textbf{LE} & y; \\ z & = & x & \$<= & y; \end{array}$$

- Equal to

$$\begin{array}{cccccc} z & = & x & == & y; \\ z & = & x & \textbf{EQ} & y; \\ z & = & x & \$== & y; \end{array}$$

- Not equal

$$\begin{array}{cccccc} z & = & x & /= & y; \\ z & = & x & \textbf{NE} & y; \\ z & = & x & \$/= & y; \end{array}$$

- Greater than or equal to

$$\begin{array}{cccc} z & = & x & >= & y; \\ z & = & x & \mathbf{GE} & y; \\ z & = & x & \$>= & y; \end{array}$$

- Greater than

$$\begin{array}{cccc} z & = & x & > & y; \\ z & = & x & \mathbf{GT} & y; \\ z & = & x & \$> & y; \end{array}$$

If the relational operator *is* preceded by a dot '.' then the result will be a matrix of l's and O's, based upon an element–by–element comparison of x and y.

- Element–by–Element less than

$$\begin{array}{cccc} z & = & x & .< & y; \\ z & = & x & \mathbf{.LT} & y; \\ z & = & x & .\$< & y; \end{array}$$

- Element–by–Element less than or equal to

$$\begin{array}{cccc} z & = & x & .<= & y; \\ z & = & x & \mathbf{.LE} & y; \\ z & = & x & .\$<= & y; \end{array}$$

- Element–by–Element equal to

$$\begin{array}{cccc} z & = & x & .== & y; \\ z & = & x & \mathbf{.EQ} & y; \\ z & = & x & .\$== & y; \end{array}$$

- Element–by–Element not equal

$$\begin{array}{cccc} z & = & x & ./= & y; \\ z & = & x & \mathbf{.NE} & y; \\ z & = & x & .\$/= & y; \end{array}$$

- Element–by–Element greater than or equal to

$$\begin{array}{cccc} z & = & x & .>= & y; \\ z & = & x & \mathbf{.GE} & y; \\ z & = & x & .\$>= & y; \end{array}$$

- Element–by–Element greater than

$$\begin{array}{cccc} z & = & x & .> & y; \\ z & = & x & \mathbf{.GT} & y; \\ z & = & x & .\$> & y; \end{array}$$

2.4 Logical Operators

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. Decisions are based on the following truth table:

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

GAUSS–386i: For complex matrices, the logical operators consider only the real part of the matrices.

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;

If the logical operator is preceded by a dot '.' the result will be a matrix of l's and 0's based upon an Element–by–Element logical comparison of x and y.

- Element–by–Element logical complement: $z =$ **.NOT** x;
- Element–by–element conjunction: $z = x$ **.AND** y;
- Element–by–Element disjunction: $z = x$ **.OR** y;
- Element–by–Element exclusive or: $z = x$ **.XOR** y;
- Element–by–Element equivalence: $z = x$ **.EQV** y;

2.5 Other Operators

- **Assignment Operator**. Assignments are done with one equal sign: $y = 3$;
- **(comma)**
  - Commas are used to delimit lists: `clear x,y,z;`
  - Commas are also used to separate row indices from column indices within brackets: `y = x[3,5];`
  - And to separate arguments of functions within parentheses: `y = momentd(x,d);`
- (period) Dots are used in brackets to signify "all rows" or "all columns": `y = x[.,5];`
- (space)
  - Spaces are used inside of index brackets to separate indices. `y = x[1 3 5,3 5 9];`.
  - Spaces are also used in **PRINT** and **LPRINT** statements to separate the separate expressions to be printed: `print x/2 2*sqrt(x);`. Therefore, no extraneous spaces are allowed within expressions in **PRINT** and **LPRINT** statements unless the expression is enclosed in parentheses: `print (x / 2) (2 * sqrt(x));`
- (colon) A colon is used within brackets to create a continuous range of indices. `y = r[1:5,.];`
- (ampersand) The (&) ampersand operator will return a pointer to a procedure (PROC) or function (FN). It is used when passing procedures or functions to other functions and for indexing procedures. See Section 4.

- String Concatenation

```
x         =  "dog";
y         =  "cat";
z         =  x $+ y;
print z;

dogcat
```

If the first argument is of type STRING, the result will be of type STRING. If the first argument is of type MATRIX, the result will be of type MATRIX. See the following examples:

```
y = 0 $+ "caterpilar";
```

The result will be a 1 by 1 matrix containing **caterpil**.

```
y = zeros(3,1) $+ "cat";
```

The result will be a 3 by 1 matrix, each element containing `cat`. If we use the y created above in the following:

```
k = y $+ "fish";
```

The result will be a 3 by 1 matrix with each element containing `catfish`. If we then use k created above:

```
t = " " $+ k[1,1];
```

The result will be a STRING containing `catfish`. If we used the same k to create z as follows: `z = "dog" $+ k[1,1];` The result z will be a string containing `dogcatfish`.

String Variable Substitution In a command like the *create fl = olsdat with x,4,2;* by default GAUSS will interpret **olsdat** as the literal name of the GAUSS data file that you want to create. It will also interpret x as the literal prefix string for the variable names **xl x2 x3 x4**.

If you want to get the data set name from a string variable, the substitution operator could be used as follows:

```
  dataset   =   "olddat";
create fl   =   ^dataset with x,4,2;
```

If you want to get the data set name from a string variable and the variable names from a character vector, use the following:

```
  dataset   =   "olddat";
   vnames   =   age, pay, sex
create fl   =   ^dataset with ^vnames,0,2;
```

The general syntax is: `^variable_name`. Expressions are not allowed.

The following commands are supported with the substitution operator in the current version:

```
      create fl   =   ^dataset with ^vnames,0,2;
      create fl   =   ^dataset using ^cadfile;
        open fl   =   ^dataset;
   output file    =   ^outfile;
        load x    =   ^datafile;
     load path    =   ^lpath x,y,z,t,v;
   loadexe buf    =   ^exefile;
    save ^name    =   x;
     save path    =   ^spath;
  dos ^cmdstr;
    run ^prog;
msym ^matring;
```

2.6 Summary

The tables below list and exemplify them compactly:

MATHEMATICAL OPERATORS		
+	addition	$z = x + y$
−	subtraction or unary minus	$z = x - y$
*	multiplication	$z = x * y$
.*	Element by element multiplication	$z = x. * y$
^	Element by element exponentiation	$x = x\hat{}2$
!	factorial	$z = x!$
./	Element by element division	$z = x./y$
/	division or linear equation solution of $Ax = b$	$x = b/A$
%	modulo division	$z = x\%3$
.*.	Kronecker product	$C = A. * .B$
*~	horizontal direct product	$C = A*\tilde{}B$
MATRIX AND STRING ARRAY OPERATORS		
—	matrix vertical concatenation	$z = x\|y$
~	matrix horizontal concatenation	$z = x\ \tilde{}\ y$
$|	string array vertical concatenation	$C = A\$\|B$
$~	string array horizontal concatenation	$C = A\$\tilde{}B$
’	transpose	$y = x'$
.’	bookkeeping transpose	$y = x.'$
STRING OPERATORS		
$+	string concatenation: A ="black"$ +"cat"	

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.

Operators, Matrix Logical The matrix 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.

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

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 example, if `x = { 0 1 0 1 }` and `y = { 1 1 0 0 }` then `(x .or y)` will be the vector { 1 1 0 1 }. Do not use the '.' operators in **if** or **do ... while** statements.

Operators, Matrix Relational The matrix-returning relational operators are:

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

The above operators all produce a matrix of 0's and 1's, with a 1 where the corresponding comparison is TRUE. The '$' is used for comparisons between character data and other nonnumeric data, e.g. NaNs.

Operators, Relational The scalar-returning relational operators are:

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

The result is a scalar 1 or 0, based upon a comparison of all elements of x and y. ALL comparisons must be true for a result of 1 (TRUE). The '$' is used for comparisons between character data and other nonnumeric data, e.g. NaNs.

Operators, Scalar Logical 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.

These operators require scalar arguments. These are the ones to use in if and do statements:

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

Conditional Branching Conditional branching is done with the if statement:

```
if x > 0;
     rv = 1;
     print "Positive";
elseif x < 0;
     rv = -1;
     print "Negative";
else;
     rv = 0;
     print "Zero";
endif;
```

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

Unconditional branching is done with goto. While 'good' programming uses a minimal of unconditional branching statements, sometimes, as in the following example, they are necessary.

```
Examples:
/* coin toss...  */
toss:
     coin = rndu(1,1);
     if coin > .49 and coin < .51;
        goto edge;
     elseif coin >= .51;
        heads = heads + 1;
     endif;
     t = t + 1;
     goto toss;
edge:
     print "It's on edge!";
     print "H" heads "T" t-heads;
```

```
/* file check...  */
open f1 = mydat for read;
if f1 == -1;
     goto errout("File not found", -1 );
endif;

errout:
      pop rv;
     pop msg;
     errorlog msg;
     _errval = rv;
end;
```

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

goto, like **gosub**, can pass arguments via the stack. If arguments are passed, they are retrieved (pop'ed) in the reverse order they are passed.

Looping Control Looping in GAUSS is a time–consuming pasttime. If you can avoid loops by programming matrices, then by all means do so. However, there are times when loops cannot be avoided. You should know that looping is controlled with the do statement.

```
  do while st > tol;                    /* loop if true */
                  .
                  .
                  .
               endo;
```

```
 do until st <= tol;                   /* loop if false */
                  .
                  .
                  .
               endo;
```

```
             break;   Jump to the statement following endo.
          continue;           Jump to the top of a do loop.
```

Creating Matrices There are a number of commands that may be used to create matrices; a list of these follows.

design	Creates a design matrix of 0's and 1's.
editm	Invokes the matrix editor.
eye	Creates an identity matrix.
let	Creates a matrix from a list of values.
ones	Creates a matrix of ones.
recserar	Computes auto-regressive recursive series.
recsercp	Computes recursive series involving products.
recserrc	Computes recursive series involving division.
seqa	Creates a vector as an additive sequence.
seqm	Creates a vector as a multiplicative sequence.
toeplitz	Computes Toeplitz matrix from column vector.
zeros	Creates a matrix of zeros.
—	Vertical concatenation operator.
~	Horizontal concatenation operator.

3 Programs

GAUSS programs can loosely be defined as files with valid GAUSS commands. A program file may consist of both the actual code of the program and additional procedures specific to that program. As a first command of a GAUSS program, the user can start to give **new** which clears the workspace. All matrices and procedures from previous programs are deleted from memory. A program can be ended with the **end**–statement, which closes all open files and terminates the program. The **pause (10)**–statement halts execution of the program for 10 seconds and the **system**–statement exits GAUSS.

An important element in any program is the flow control. Various keywords are available in GAUSS to determine whether a piece of code should be repeated some times or should be executed at all. A GAUSS–loop is started using the do-while statement and ended by the endo statement. Within the loop, one can jump to the top of the loop with the **continue**–statement and one can break out of the loop with the **break**–statement. In that case the program proceeds with the first command following endo. Consider the following example:

```
i=l;
do while (i<=100);
i=i+l;
print "i=" i;
endo;
```

Note that a Boolean expression `i<=100` determines whether the code within the loop should be executed again or not. Alternatively, the loop can be controlled using the **do until** statement as in

```
i=l;
do until (i>100);
i=i+l;
print "i=" i;
endo;
```

It is not advisable to perform matrix operations using a *do–while* loop if they can be performed using elementwise operators instead. The following program

generates 10000 random numbers uniformly distributed between 0 and 10 and classifies them into the intervals [0, 3), [3,7), and [7, 10]. (**hsec** is a standard GAUSS function that returns time elapsed since midnight in hundredths of a second.)

```
r=10000;
v=10*rndu(r,1);

etl=hsec;
vl=zeros(r,1);
i=1;
do while (i<=r);
if v[i]<3;
vl[i]=1;
elseif v[i]>7;
vl[i]=3;
else;
vl[i]=2;
endif;
i=i+1;
endo;
etl=(hsec-etl)/100;
et2=hsec;
v2=(v.<3) + 2*(v.>=3).*(v.<=7) + 3*(v.>7);
et2=(hsec-et2)/100;

print "loop" etl;
print "vectorized" et2;
print "ratio" (et2/etl);
```

Classification is much faster if done using the elementwise operators, in this particular example the 'vectorized'-code is almost three times as fast. In fact, even faster code is

```
v2=1+(v.>=3)+(v.>=7);
```

Code is executed conditionally using the **if ... endif** statements. Within an **if ... endif** branch one can use **elseif** and **else** statements for further conditioning. Both the **if** and **elseif** statements must be followed by a scalar expression which determines whether the code should be executed or not. Each **if** statement must be ended with an **endif** statement. Consider the following example:

```
j=1;
do while (i<=20);
if (i%2==0);
print lli is even" i;
elseif (i%3==0);
print "i is odd and divisible by 3" i; else;
print "i is odd and not divisible by 3" i; endif;
i=i+1;
endo;
```

After writing a program in the GAUSS-editor, one can compile and run the program by clicking on the run button.

4 GAUSS Procedures

Procedures are the building blocks of GAUSS. Many useful procedures are provided with the installation and other handy procedures are shipped with the GAUSS modules or with commercial extensions to GAUSS. GAUSS derives its flexibility from the possibilities for the user to write his own procedures. These procedures can be very simple or complex, even though it is recommendable to break complex procedures in a few simpler ones. In this section we will discuss writing a procedure and some important standard procedures that belong to the standard installation of GAUSS.

A procedure is created along the following steps. First, the source code must be written in a file. That file must have the same name as the **procedure** , and have extension **g**. For example, the code of the procedure *boxcox* must be in the source file **boxcox.g**. This requirement implies that a procedure name can have up to eight characters. The file with the source code must be placed in a subdirectory that is listed in the path for program files (the variable *src–path* in *gauss.cfg*). The actual code for the procedure consists of the following five parts:

1.	procedure declaration	`proc(`*x*`) pname(`*agruments*`);`
2.	declaration of the local variables	`local`*list*
3.	actual code of the procedure	*GAUSS statements*
4.	returning values	`retp(`*list*`);`
5.	end of the procedure	`endp;`

It is good practice to document the most important features of the procedure in the first couple of lines, between the comment terminators /∗ and ∗/. If the user needs help on the procedure he can use the GAUSS help system as if the procedure were an intrinsic command or a vendor provided procedure. The help system puts the source code of the procedure in the GAUSS–editor, hence documentation should be in the top of the source file.

One should note that GAUSS is not a very 'safe' language in the sense that it hardly performs any type checking at either compilation or run time. Hence, the user should do this when he writes the procedure.

Every procedure starts with a declaration, as for example:

```
proc (1) = boxcox(x,b);
```

The number of returns is put between parentheses. If only one object is returned this can be left out as in `proc boxcox(x,b);`. The parameters of the procedure are passed after the procedure name. In this case, there are two parameters: x and b.

The second element of the procedure is the declaration of the local variables. All variables in GAUSS are global ones, unless they are preceded by the keyword local when they are declared. All global variables are accessible from within the procedure and are not declared in one way or another. There is much that could be said as to the advantage or disadvantage of using global variables in a procedure. Good practice would be to use only local variables but in an iterative context, this may be time consuming as the local variables must be assiged at

each call to the procedure.

If the procedure is specific to one program (for example, a procedure that calculates a specific likelihood function to be optimized) the code of the procedure may also be placed in the file that contains the code of the GAUSS program. In this case, that procedure can not be called from other programs. If the procedure is part of a library, this is not necessary. Creation of libraries is discussed in Section 5. However, if the procedures are to be placed in a library, care must be taken if the routines contain global variables.

Local variables are declared as in

```
local z;
```

After this command, z must be initialized as it is not initialized by its declaration. A local variable may have the same name as a global variable, within the procedure where it has been declared as *local*, the local variable temporarily 'overrides' the global variable of the same name. All parameters are passed by reference.

The third part of the procedure is generally the most interesting part. Here the actual calculations are performed. In this section other procedures may be called.

The fourth element of the procedure is returning the result of the calculations. The number of elements returned must coincide with the number of returns given in the declaration. An example of the return statement is

```
retp( z );
```

If the procedure has no returns, this part can be skipped; if more than one element is returned, the elements are separated by comma's as in

```
retp(z,x);
```

Finally, every procedure is terminated with the

```
endp;
```

command. An example of a procedure that calculates the Box-Cox transform (the Box-Cox transform is a transformation in statistics that transforms a variable according to $x^{(\lambda)} = \frac{x^{\lambda}-1}{\lambda}$ if $\lambda \neq 0$ and $x^{(\lambda)} = \ln x$ if $\lambda = 0$

```
proc (1)=boxcoxl(x,lam);
/*
input x:  n x k matrix with positive elements
lam:  scalar

output:  n x k matrix with Box Cox tranformation of each element of x
/*

if (rows(lam)/=1 or cols(lam)/=1);
   errorlog "boxcoxl.g:  lam must be a scalar";
   retp(-1);
endif;
if (lam==0);
   retp(ln(x));
else;
   retp(x.^lam-1)/lam;
endif;
endp;
```

This procedure does two things. First it checks whether the second argument, `lam`, is a scalar. If this is not the case, an error message is printed using the **errorlog** command and the procedure returns value -1. After this check, the actual calculations, which depend on the value of the first argument x are performed. Note that no local variables are declared to minimize memory requirements. Moreover, note that it is possible to exit from the procedure with a **retp**–statement from anywhere within the procedure.

Procedures can be called in different ways (*eigrs2* is a procedure that calculates the eigenvalues and eigenvectors of a real, symmetric matrix):

```
x=boxcox(y,0.5);

boxcox(y,0.5);

{va,ve}=eigrs2(h);

call eigrs2(h);
```

In the first case, the result of the procedure is stored in the matrix x. This matrix x is created and initialized automatically, if necessary. In the second case, the output of the procedure **boxcox** is copied to screen. The output of the procedure **eigrs2** consists of two elements. The first matrix in the **retp**–statement of **eigrs2** is stored in *va* and the second matrix in that statement in *ve*. In the fourth case, the procedure is executed and all output is discarded. This way of calling a procedure may be useful if the return indicates successful completion of the procedure only (as is usually the case with, for example, the procedure **xy** which does a two–dimensional graph) and one is not interested in saving the result.

Sometimes a procedure uses global variables (for example the ols- and dstat-procedures to be discussed below). In general, it is better to pass global variables as parameters to the procedure. This makes the procedure more usable in another context: global variables tend to be there when you need them, but they also tend to show up when you don't expect them. The following program is valid GAUSS code; it shows the dangers of accessing global variables from

within a procedure.

```
new;
a=3;
change_a;
print a;

proc (0) = change_a();
a=5;
endp;
```

The printed result is 5 even though most users would expect the result 3 from the `print a`.

A special kind of argument of a procedure is a pointer to another procedure. This is useful when writing some general purpose routines that take a procedure as their argument. For instance, the derivative of a procedure can be approximated by

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

with h small number. This general purpose procedure to approximate a derivative can be programmed taking a pointer to the function f as its argument. See the example below:

```
x=1|2|3;
numder(&x2,x,1e-8);

proc numder(&f,x,h);
/* approximates the derivative of f at x */
local f:proc;
retp((f(x+h)-f(x-h))/(2*h));
endp;

proc x2(x);
retp(x^2);
endp;
```

Pointers to these procedures are obtained by preceding the name of the procedure with an ampersand (&), a convention well–known to C and C++ programmers. Note that the symbol for the procedure to be passed in the argument list of the procedure *numder* (f in this case) must be declared as a local procedure in the local declaration list (`local f:proc`). It is the responsibility of the user that f is called correctly within *numder*. For instance, if the procedure x2 were redefined as

```
proc x2(x,k);
retp(x^k);
endp;
```

GAUSS would give an error message because the call $f(x+h)$ in the procedure *numder* would be invalid, as this procedure has one parameter only. A more complex example is the program below where a procedure is passed as a pointer to another procedure in order to estimate a density by kernel estimation:

```
/*
   program to illustrate procedures as
   arguments to other procedures
/*

new;
library pgraph;
nobs=10000;
ngrid=500;
rndseed 67;
Y=sortc(rndn(nobs,1),1);
z=seqa(-3,6/ngrid,ngrid);

fl=kernelestimate(z,y,0.1,&biweight);
f2=kernelestimate(z,y,0.1,&gaussian);
ftrue=pdfn(z);
_pltype = 6;
_pcolor =  11, 12, 14 ;
call xy(z,fl~f2~ftrue);

proc kernelestimate(z,x,h,&kf);
local arg,i,f,kf:proc;
i=1;
f=zeros(rows(z),1);
do while (i<=rows(z));
arg=(z[i]-x) /h;
f[i]=meanc(kf(arg))/h;
i=i+1;
endo;
retp( f);
endp;

proc biweight(x);
retp( (15/16*(1-x^2)^2).*(abs(x).<1));
endp;

proc gaussian(x);
retp( pdfn(x));
endp;
```

In this example, pointers to the *biweight-* and *gaussian*–procedures are passed as parameters to the procedure *kernelestimate*.

GAUSS is shipped with many standard procedures. Two important procedures are the procedure to perform linear regression and the procedure to calculate descriptive statistics. Both procedures use both local and global variables. First we discuss the regression procedure. The procedure estimates parameters in the linear model, where $i = 1, \ldots n$:

$$y_i = \beta' x_i + \epsilon_i \tag{1}$$

In equation (1), the k vector x_i is the vector with regressors that may or may not include a constant term. The disturbances ϵ_i are assumed to be independently distributed with zero mean and (constant) variance σ^2. The parameters of the model (β and σ^2) can be estimated by

```
{vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat}=ols(dataset,depvar,indvar);
```

The parameters of this procedure are *dataset* (a string variable containing the name of the dataset), *depvar* (a character vector with one element or a scalar pointing to the row in the dataset with the dependent variable) and *indvar* (a character vector with the names of the independent variables or a vector with the row indices of the independent variables). If dataset is a null string, the actual vector of the dependent variable and the matrix with independent variables are assumed to be passed to the procedure as *depvar* and *indvar*. The following two calls yield the same results:

```
call ols("testols","Y1","X1"|"X2");
call ols(0,y,xl~x2);
```

assuming that the dataset *testols* contains the same values for the variables as the vectors y, xl, and $x2$. The way OLS estimates are calculated is partly determined by global variables. These global variables are also known as 'flags'.

__con	if this variable is set to 0 the regression is estimated without an intercept. The default value is 1 so that an intercept is included.
__miss	this one determines how missing values are treated. The default value is 0 so that the procedure assumes all observations are valid.
_olsres	If this variable is set to 1, the Durbin-Watson test statistic for autocorrelation is calculated and the OLS-residuals are determined. The default value is 0.
__altnam	This may be set to a character vector with the names of the variables, with the name of the dependent variable as the last element.

Detailed information on this procedure can be obtained from the online help.

Another useful standard procedure is **dstat**. This procedure calculates the mean, standard deviation, minimum, maximum and number of valid cases of a dataset or a datamatrix. Its syntax is

```
{vnam,mean,var,std,min,max,valid,missing}=dstat(dataset,vars);
```

Again, *dataset* is a string variable with the name of the dataset to be analyzed and *vars* may be either a character vector or an index vector. If vars is 0, descriptive statistics of all variables in the dataset are listed. If *dataset* is 0, *vars* is assumed to be a data matrix that is analyzed. Treatment of missing values is determined by the global variable *__miss* described above. An example where both the **ols**- and **dstat**–procedure are used is

```
nobs=100;
x=ones(nobs,1)~3*rndn(nobs,3);
beta={ 0,1,-1,0.5 };
sigma=1;
y=x*beta+sigma*rndn(nobs,1);

_olsres=1;
--altnam="CONSTANT"|"X1"|"X2"|"X3"|"DEPVAR";
call dstat(0,x~y);
call ols(0,y,x);
```

5 Libraries

GAUSS is a modular language, it can be extended by libraries that add new procedures to the language. These libraries are distributed both as commercial add–ons to GAUSS as well as free software. In this section we will describe how add–on libraries can be used as well as how one can write ones own library.

Before we discuss writing libraries, we need to discuss the way GAUSS searches for unknown references (like a matrix or a procedure). If the compiler encounters an unknown object during compilation, the state of the autoloader determines how the program searches to resolve the unknown reference if the reference cannot be resolved in the file under compilation. First, the *autoloader* searches in the current directory (which is listed in the lower-right corner of the screen) and then along the *src*–path given by *src_path* in the file gauss.cfg. The exact way how is searched for unknown references is determined by the state of the autoloader and the autodelete state found in the *options–program*–meny.

Suppose GAUSS encounters this line of code: `{b,s2}=ols_estimate(y,x);` Since `ols_estimate` is not an intrinsic GAUSS function, code for this function needs to be located and compiled. If the *autoloader* is turned off (that is, the box in the meny is not checked), then the procedure must have been declared before with the command

```
external proc ols_estimate;
```

In that case forward references (ie, references to objects not already defined) are not allowed. If both the *autoloader* is on and the *autodelete–state* is on GAUSS searches for the unknown object along the following paths. First, GAUSS tries to find it in the user library, then in user-specified libraries, then in the GAUSS library and finally it searches for files with a ".g" extension in the current directory and along the path listed in src–path.

In case the *autoloader* is on but the *autodelete–state* is off, GAUSS does not search for files with a '.g' extension. Moreover, forward references to objects not listed in the libraries are not allowed.

Compilation time is longest when both the autoloader and the autodelete-state are on, but that case is most convenient to the user. In the remainder of this section we assume that both are on.

A GAUSS library is best thought of as an index file where the program can find the exact location of references. Libraries are activated by a command like

```
library maxlik, bstat;
```

After activation, procedures and matrices defined in these libraries become available in a program. Two libraries are activated by default when GAUSS is started: the gauss- and user-libraries. The gauss library is a library with native gauss procedures like *dstat*, *bstat* and many others. The user–library is empty at the moment GAUSS is installed on ones system, but the user can add his own procedures to this library (see below). When a GAUSS–session is started, it is examined whether a user–library exists. If this is not the case the user is prompted if such a library must be created.

GAUSS libraries are stored as ASCII-files with extension '.lcg' in the di-

rectory specified by the variable *lib*–path in gauss.cfg. An example of such a library file is the one below. The file would be an ASCII one contining three rows (indentation is optional) and might be called *course.lcg*:

```
f:\prog\testlibl.src
   ols_estimation      :   proc
   ml_estimation       :   proc
```

This library consists of two objects: two procedures (`ols_estimation` and `ml_estimation`). If the course–library has been activated by the command

```
library course;
```

and the compiler encounters a reference to either one of these objects, GAUSS 'knows' where to find the code for that particular object and the file containing that object (in this example testlibl.src) will be compiled. In fact, all code in testlibl. src will be compiled so it is sensible not to create excessively large library source code files. It is not necessary to have all code for one particular library in one file, as the following example shows:

```
/*
** tscs.lcg
** Time Series/Cross Sectional Analysis Library
** (C) Copyright 1988-1996 by Aptech Systems, Inc.
**All Rights Reserved.
*/
tscs.dec
     _ts_ver                                          :   matrix
     _tsmodel                                         :   matrix
     _tsstnd                                          :   matrix
     _tsmeth                                          :   matrix
     _tsise                                           :   matrix
     _tsmnsfn                                         :   string
     _ts_mn                                           :   string

tscs.src
     tscs                                             :   proc
     _tsgrpmeans                                      :   proc
     _tsprtp                                          :   proc
     _tscsset                                         :   proc
     _tsfile                                          :   proc
```

This library is the library file of the time-series/cross-section library, one of the libraries commercially available. All code for this library is found in two files: tscs.dec (a file with declarations of global variables, see below) and tscs. src (a file with the actual procedures that make up the library). Usually library files with source code have extension .src or .arc.

A second file with source code (say, testlib2.src) is added to our course–library by

```
lib   course   testlib2.src
```

at the GAUSS–command prompt. The list of active libraries can be found by giving the command library without any library names.

Most libraries use global variables to allow the user to determine how cal-

culations are made. These global variables can be declared and initialized at compile time using the *declare*–statement. In the following source code file, the OLS–estimator is calculated, and an estimate for the variance of the error term in the linear regression model.

```
#include testlib3.dec

proc (0)=testlibset;
_df_correction=1;
endp;

proc (2)=ols_estimation(y,x); /*
Does an OLS estimation

input:  y dependent variable
       x matrix of independet variables

output:  b vector of estimated coefficients
        s2 residual variance
*/
local n,k,b,e,s2;
n=rows(x);
k=cols(x);
b=y/x;
e=y-x*b;
if (_df_correction==1);
s2=e'e/(n-k);
else;
s2=e'e/n;
endif;
retp( b, s2);
endp;
```

When this file is compiled, the *include*—statement 'reads' the file testlib3.dec into testlib3.src and the resulting code is compiled. The file testlib3.dec contains the line

```
declare _df_correction?=0;
```

Here, the global variable *_df_correction* is declared and initialized. When the code of the procedure ols_estimation is compiled, GAUSS 'knows' that *_df_correction* is a global variable with value 0, unless the user has initialized this variable earlier. In that case, *_df_correction* is not reinitialized, because it is declared using ?=. If the usual = were used, *_df_correction* would be reinitialized, even if it were initialized before. For details of initializing variables we refer to the manual.

Complementary to the declaration file we have the file testlib3.ext:

```
external matrix _df_correction;
```

The external-command defines a variable without initializing it.

6 File input–output

One of the more troublesome issues with any computer language is corresponding with other programs. GAUSS can read ASCII-datafiles and it reads and writes datafiles in a proprietary binary format that is not recognized by other programs. It is not possible to import binary data files of well–known programs as SPSS or LOTUS without the help of third-party applications.[4]

An ASCII-file with numeric data is read into a vector x with the **load**–statement: `load x[]=file.asc;`. The data are read row–wise from file.asc and must be separated by a white space (ie, a space, tab or return). The data in file.asc must be data suitable for storage in the GAUSS data type matrix: they must be numerical or a sequence of characters starting with the character 'a', . . . 'z' or their uppercase equivalents. GAUSS stops processing the input stream as soon as a nonnumerical entry with a nonvalid first character is found (for example &) even though valid data may follow.

Compiled procedures, strings, and matrices can all be saved in a binary format. Most often, one wants to save a matrix or a string (array) so that these can be used later for further analysis. The general format of the **save**–command is save *name* = *symbol*; where *name* is a literal or a referenced string and *symbol* is a symbol (a matrix or a string, for example). Examples are (x is a matrix and s is a string):

```
save x;
save names=s;
save c:\tmp\xx=x;
save path=c:\tmp x;
```

In the first case, x is saved into the binary file *x.fmt* in the current directory, in the second case, s is saved into tile binary file *names.fst* in the current directory, in the third case, x is saved into the file *xx.fmt* in the directory c:\tmp and in the final case x is saved into the file *x.fmt* in the directory c:\tmp and all further objects saved with the **save**–command will be placed in this directory, unless an explicit filename is given as in the third example. The extension '.fmt' indicates a matrix written to disk and the extension '.fst' indicates a string written to disk.

A matrix file created using the **save**–command can be loaded into the workspace using the *load*–command, for example by *load x* (x.fmt is loaded from the current directory into x) or load x=c:\tmp\xx (c:\tmp\xx.fmt is loaded into x). A binary file containing a string can be loaded with the **loads**–command, which works analogously to the **load**–command. Note that the **load**—command is used both to read ASCII-data into a GAUSS matrix and to read matrix files saved earlier in a GAUSS-session. It is not possible to read ASCII-data using the **loads**–statement.

Sometimes it is convenient to save data in ASCII–format, instead of in the GAUSS binary format, for instance, if the data are to be used in another program. This is accomplished by printing the data to a file. First, one defines an output file using the **output**–command. Then the data are printed, and finally the output file is closed. An example is

[4] Newer versions of GAUSS will import *MS Excel* files.

```
output   file=test.out   reset;
print x;
output off;
```

Here, the matrix x is printed to the file test.out. Note the **reset**–modifier in the output statement. This causes an existing test.out file to be overwritten. The modifier is required if the file does not exist. Formatting of entries of the matrix can be changed if necessary using the **format**–command.

It is not always practical to store data in a matrix. Sometimes it is more convenient to store them in a dataset on disk, especially if the dataset is large. A dataset can be created in two different manners: from within GAUSS or by using a conversion utility that converts a dataset in ASCII format into GAUSS format.

First, one can use the **saved**–command as in *saved(x,dataset,vnames)* where x is the matrix to be saved in the datafile, *dataset* is a string variable containing the name of the datafile and *vnames* is a vector with the names of the columns of x. Upon successful completion saved returns the value 1. It is customary to assign variable names in capitals for numerical variables and in lowercase for character variables. If no vector with variable names is passed (i.e., *vnames* is set to 0), GAUSS creates a vector with variable names automatically, with names Xl, X2, etc.

```
/* illustration of writing and reading datasets */

x=3*rndn(1000,5);
y=rndn(1000,1);
c=(y.<0).*"ltzero"+(y.>=0).*"gtzero";
data=x c;
/* uppercase:  numerical variables */
/* lowercase:  character variables */
vnames="Xl"|"X2"|"X3"|"sign";
file_name="testdatall";
saved(data,file_name,vnames);

z=loadd(file-name);
```

Data can be read into memory using the **loadd**–command as in the example above. This, however, is possible for small datasets only. If the dataset is large data can be read by reading successive chunks from the datafile. Alternatively, data can be read row–wise from a dataset, as in the example below.

```
/* illustration of reading datasets row-wise */
file_name="testdata"; vnames=getname(file_name);
i=1;
data={};
open fh=^file_name;
number_rows=rowsf(fh);
do while i<=number_rows;
data_row=readr(fh,1);
data=data|data_row;
i=i+l;
endo;
close(fh);
```

First, one assigns a file handle to the file that is going to be read. Then

Table 1: Graph types available in GAUSS.

Graph types			
xy(x,y)	logx(x,y)	logy(x,y)	loglog(x,y)
bar(v,h)	hist(x,v)	histf(x,f)	histp(x,p)
box(g,h)	xyz(x,y,z)	surface(x,y,z)	contour(x,y,z)

the dataset is accessed using the *readr(fh,1)*–command. The first argument is the file handle, the second argument is the number of rows that is going to be read. Finally, the dataset is closed. All open files in a GAUSS session may be closed with **closeall**. The variable names of the columns in the dataset can be retrieved by the command *vnames=getnames(dataset)* where *dataset* is a string variable with the name of the dataset.

7 Graphics

GAUSS is equipped with some graphics capabilities. With a few commands one is able to plot two- and three dimensional data and the graphs can be saved for incorporation in a word processor. In the first subsection we will deal with the most important commands of the graphics library, in the second subsection we show how to include GAUSS graphics in a LaTeXdocument.

7.1 Creating graphics in GAUSS

In this section we discuss configuration of the graphics library and the most important commands. The graphics library of GAUSS is configured analogously to the way GAUSS is configured by editing the file *gaussi.cfg*. The configuration settings for the graphics library are stored in this file. Some of the options set in this file can be changed while running GAUSS (all print options), but the video adapter type is determined by MS Windows. GAUSS supports various printers also through the MS Windows interface. For DOS users, other rules apply. Also, for these users, it is important that the directory that contains the file gaussi.exe is listed in the DOS search path. It will not be possible to run the graphics program if that is not the case. General help on the graphics library is to be found in the on–line help function.

The graphics library is activated by library pgraph and global variables are set to their default values by calling the procedure **graphset** (a procedure without parameters). GAUSS graphics are created in four steps:

1. Activation of the graphics library.
2. Reading and formatting of the data.
3. Setting global variables to special values.
4. Calling the actual graphics procedure needed.

Here we focus on the third and fourth step. After a graph is displayed on screen the user has two options. Pressing **¡ESC¿** lets the program continue and pressing ↵ yields a menu with some saving and printing options. The above table contains the commands that generate graphics. The names of the procedures indicate the type of graph it creates. The most useful graphs that can be generated are two– and three–dimensional plots (**xy** and **xyz**), contour and surface plots of three–dimensional data (**contour** and **surface**), and histograms (**hist**, **histf**, and **histp**). To illustrate the use of the graphics library, we give a simple example first. A graph of the standard normal density function is generated in the following program:

```
/* density of standard normal */
library pgraph;
graphset;
grid_size=.05;
x=seqa(-3,grid_size,6/grid_size);
y=pdfn(x);
xy(x,y);
```

The standard normal density function is evaluated on a grid of the horizontal axis; the distance between successive points is 0.05. If the distance would be too large, say 1.0, the graph would be too jagged. Output is partly determined by global variables of the graphics library. Of course, these values must be set before the graph is drawn using one of the drawing commands. Global variables are set by calling a procedure (for example, `title("example plot")`) or by explicit assignment (for example, *_pdate=O*). Not all global variables are applicable to each graph type. Global variables are reset to their default values by calling the procedure *graphset*. Axis can be named with `xlabel("text on x--axis")`, `ylabel("text on y-axis")`, and `zlabel("text on z-axis")` and a title is given with `title(''title of the plot'')`. The date and time of creation of the graph is printed in the upper left corner. This feature can be turned off by setting the global variable _pdate=O;. If _pdate is set to a string, then this string will be printed in the upper left corner and the date will be appended.

The commands *hist(x,v)*, *histf(f,c)* and *histp(x,v)* in Table 1 all produce histograms. They differ only in type of information displayed on the vertical axis (absolute numbers, frequencies of percentages) and input requirements (raw data and a vector of breakpoints (*hist* and *histp*) or frequencies and a vector with category labels (*histf*). In the example file below we generate a vector with normally distributed numbers and draw a histogram using {-1.5, -1, 0, 1, 1.5} as breakpoints so that the first bar corresponds to observations $x < -1.5$, the second category to $-1.5 < x < 1$, etc.

```
library pgraph;
graphset;
x=rndn(100,1);
b={-1.5, -1, 0, 1, 1.5};
title("histogram of normal distribution");
ylabel("frequency");
xlabel("x value");
_pdate=0;
call hist(x,b);
```

Two dimensional graphs can be plotted using the *xy(x,y)*–command. Of course, the arguments x and y must be of equal length unless one wants to index the observations. The command *xy(1,y)* plots the curve $(1, y_l)$, $(2, y_2)$, etc. The second argument need not be a vector: if a matrix (with the same number of rows as x) is passed as an argument, each column will be graphed. Consider for example the following program

```
library pgraph;
graphset;
ngrid=100;
x=seqa(-3,6/ngrid,ngrid);
y1=pdfn(x);
y2=1/pi*1./(1+xmyhat2);
_plegctl=1~4~1~.32;
_plegstr="Gaussian density\000Cauchy density";
xlabel("ordinate");
ylabel("density");
call xy(x,yl~y2);
```

In this example, two density functions are graphed, viz. the density of the normal distribution and the density function of the Cauchy distribution. The second argument passed to the **xy**–procedure is a 100 by 2-matrix and each column is graphed as a separate line. The legend in the graph is set by two variables. The 4-row vector *_plegctl* determines where the legend is placed. The first element determines whether the coordinates are set in plot coordinates (1), inches (2), or pixels (3). The second element gives the font size of the legend (between 1 and 9) and the final two elements give the horizontal and vertical coordinate of the lower left corner of the legend (in units specified by the first argument). In our example, coordinates are given in plot coordinates, font size is 4 and the coordinate of the lower left corner is (1, 0.32). The text of the legend is specified in *_legstr*. Different curve labels are separated by the ASCII-0 character \000. The label of the first curve is 'Gaussian density' and the label of the second curve is 'Cauchy density'. If the number of labels given in *_plegstr* is less than the number of columns in y empty labels will be printed in the legend otherwise the order of the curves in the legend correspond to the order of the columns in y, Additional text messages can be ploted in the graph using the global variables *_pmsgctl* and *_pmsgstr*. By its default settings set using the graphset procedure, a line is drawn through the points plotted. The type of line plotted is controlled by the global variable *_plctrl*. If this variable is set to -1, only individual points are plotted and these points are not joined by a line. The symbol at each point is controlled using *_pstype*.

It is also possible to draw three dimensional curves and contour lines are graphed using the surface and contour procedures stated in Table 1. Both procedures have three arguments: a row–vector x of length k, a vector y of length p and a p x k matrix z. The rows of z correspond to the elements of y and the columns of z correspond to the elements of x. A graph of the surface above the (x, y) plane is drawn by *surface(x,y,z)* and a two dimensional graph with isocontour curves is drawn by the command *contour(x,y,z)*. A three dimensional graph of a function is obtained with the *xyz(x, y, z)*–procedure.

This procedure yields 3D results analogous to the xy procedure discussed

above. As an example we give the following program that generates three 3D plots based on a bivariate normal density function:

```
library pgraph;
graphset;

ngrid=51;
x=seqa(-3,6/ngrid,ngrid);
y=x;
x1=pdfn(x);
y1=pdfn(y);
z=x1.*y1';

xyz(x,y,z[.,1:2]);
surface(x1,y,z);
contour(x1,y,z);
```

Graphical images can be saved to disk in one of the file formats supported using the graphtprt procedure. The argument of this procedure is a string that contains some flags. The most important flags are -p (print graph using the settings specified in pqgrun.cf g), -pf=test (print output to a file test), -c=k (convert file format to (k=1) Encapsulated Postscript, (k=2) Lotus PIC-file, (k=3) HPGL format or (k=4) PCX colored bitmap format) and -cf=test (print output to a file called test). For example, the first graph drawn after the command *graphprt("-p -pf=test")* will be printed automatically to a disk file test using the printer default in pqgrun.cfg. Of course, the picture must be redrawn before these settings take effect. This can be done explicitly by calling the graphing procedure of choice, but also using the rerun command. In the latter case, the last graph is redrawn.

7.2 Incorporating GAUSS graphics in documents

I have have bad experience in trying to print GAUSS graphics after saving them in different formats. If one is doing LaTeX, then I would strongly suggest that one save the *data* that is used in the plot and do the actual drawing in another program such as *Axum* — which I use — or *CoralDraw*. Even *Excel* can be used for most graphs.

If one is using *Word for Windows* or some other windows based program, then I have found that conversion to the HPGL format and subsequent importing to the document is the best way to go. However, my experience here has been with the DOS version: it seems that the windows version will not export graphics at the moment. At least not for me.

8 Doing research wiith GAUSS

While GAUSS is fun to play around with, it is a powerful program that can be, and indeed is, used in different research applications: simulating non–linear systems and the maximum likelihood estimation are but two examples of applications of the program. In this concluding section, I will talk about the latter in the context of estimating a *Logit* regression.

The basic model is

$$y_i^\star = x_i\hat{\beta}$$

However, the dependent variable, $y_i^\star$ is not observable. Instead we observe is $y_i = 1$ if $y_i^\star > 0$ and $y_i = 0$ otherwise. Note that $x_i\beta$ is not $E(y_i|x_i)$ as in the linear probability model but rather $E(y_i^\star|x_i)$; this implies that the residual is not available when we estimate models based on y_i. This was the specification error in Edward's article referred to above. This means the probability that $y_i = 1$ is

$$P(y_i = 1) = P(u_i > -x_i\beta) = 1 - F(-x_i\beta)$$

where F is the cumulative distribution function for the u_i's.

Note that the values of y_i are realizations of a binomial distribution with the probabilities as above. Thus the likelihood function is

$$\begin{aligned} L &= \prod_{y_i=0} F(-x_i\beta) \cdot \prod_{y_i=1} [1 - F(-x_i\beta)] \\ &= \prod_i [F(-x_i\beta)]^{1-y_i} \cdot [1 - F(-x_i\beta)]^{y_i} \end{aligned}$$

The exact form of the likelihood function will depend upon the assumptions on the distribution function F. In practice, we maximize the natural log[5] of this function:

$$\ln L = \sum_i (y_i \ln [1 - F(-x_i\beta)] + (1 - y_i) \ln [F(-x_i\beta)])$$

Note that an equivalent specification, and the one we shall use, is found in Amemiya, *Advanced Econometrics*, p. 271). Here I write it as the log of the likelihood function *for a single observation*:

$$\ln L_i = (y_i \ln [F(x_i\beta)] + (1 - y_i) \ln [1 - F(x_i\beta)]) \tag{2}$$

Let us now spepcify the density of the logistic distribution:

$$F(x_i\beta) = \frac{\exp(x_i\beta)}{1 + \exp(x_i\beta)} \tag{3}$$

This makes for an especially simple frequency function:

$$f(x_i\beta) = F(x_i\beta) \cdot [1 - F(x_i\beta)] \tag{4}$$

The likelihood function (2) is maximized by setting its first partial derivatives to zero:

Thus the first order conditions obtain by setting the gradient vector of the log likelihood function equal to zero:

$$\frac{\partial \ln L_i}{\partial \beta} = (y_i - F(x_i\beta))\, x_i = 0 \tag{5}$$

[5]Unless explicitly stated otherwise, all logarithms are to be understood as the natural logarithms of base e

The Hessian is for the second order conditions is always negative definite:

$$\frac{\partial^2 \ln L}{\partial \beta \partial \beta'} = -\sum_i F(x_i\beta)\left(1 - F(x_i\beta)\right) x_i x_i' \tag{6}$$

Given equations (2), (5) and (6), with the Logit distribution understood to be the F function, we now have the necessary information to begin a brief discussion of maximum likelihood.

8.1 Maximizing a likelihood function

All maximizing routines have a number of features in common:

1. It goes without saying that the first requirement is a function to be maximized along with the data — if any — to be used.

2. A second requirement is strarting values for the unkown parameter vector. The routine will begin here and try to find better values for these unknowns. Given the initial values, the program calculates the gradient vector at the starting point as well as the a matrix of second derivatives or an appriximation of this matrix. Just how this matrix is calculated depends upon the method used. The Newton-Raphson method uses the Hessian; the method of scoring uses the expected value of the Hessian; BHHH uses the outer product of the gradient vector. If analytical routines for the these matrices are not available, numerical approximations are used and the maximizing algorithm employed will be the Daviddon–Fletcher–Powell or a variant thereof.

   After an iteration, the parameter vector is given new values. The updating formula is

$$b_{i+1} = b_i - sH^{-1}g \tag{7}$$

   where H is the Hessian and g is the gradient of the function under consideration. s is the step size. If this is too large, then we overshoot; if it is too small then we do to many iterations. There are different ways to calculate the size of s but a simple method usually works quite well, especially if the objective function is strictly concave. For example, one method begins using the full step ($s = 1$) and stops when the step is just large enough to increase the function value. Note that this is rather *ad hoc* but it works.

3. A convergence criterium will the the routine when it has found the answer: if the gradient is sufficiently close to zero, then we have found the maximum and the iterations stop. Note that if there are any local maximum points, the routine may converge to one of these rather than the global answer.

4. Finally the routine should provide an estimate of the standard errors to the parameters.

The routine **m**axlik is a routine in GAUSS that will perfor these calculations.[6] This routine is called using

```
{par,f,g,H,retcode} = maxlik(data,var,&likeli,start)
```

The arguments are in this call are as follows:

The arguments in the call	
`data`	the data
`var`	the list of variables in the data that will be used: this is set to zero if all the data is being used
`&likeli`	the name of the function to be maximized. More will be said about this below. Here we should note that the routine should calculate the loglikelihood for each observation.
`start`	the initial values
The 5 variables on output	
`par`	the estimated values of the parameters at conversion
`f`	the average value of the final liklihood
`g`	the derivative of the entire likelihood function at conversion
`H`	the Hessian matrix at conversion
`retcode`	A variable describing why the routine stoped. `retcode = 0` at normal convergence. Other values indicate some problem was incountered adn the routine stopped before the convergence criteria were satisified.

There are innummerable global variables that control the flow of the routine. There are a few that I list in the following table:

[6] I describe version 4 of this program.

_max_Algorithm	This allows the used to choose the maximizing algorithm: '1=steepest decent', '2=BFGS', '3=DFP','4=Newton–Raphson', '5=BHHH' and '6=PR conjungate gradient'. With numerical derivatives, use 2 or 3; with analytical first derivatives use 2, 3 or 5; with analytical first and second derivatives, use 4.
_max_CovPar	A scalar to set the type of covariance matrix of the parameters that is calculated: '0=inverse of the final Hessian', '2=the inverse of the second derivatives', '3=Outer Product Gradient', '4=Heteroscedastic–consistent CVC' $(H^{-1}(g'g)H^{-1})$.
_max_GradProc	A pointer to the routine to calculallte first derivatives. This is optional.
_max_HessProc	A pointer to the routine to calculate the Hessian matrix; this too is optional.
_max_IterData	A 3 by 1 vector contin, on output, the number of iteration, the run time in minutes and a code indicating the type of covariance matrix calculated.
_max_LineSearch	This controls the methos used for the step size mentioned in (7). '1=unit step length', '2=cubic or quadratic step length' (the default), '3=step halving', '4=Brent', and '5=BHHH step length'.
_max_ParNames	A vector with the names of the parameters.

8.2 An example

All that is left is to specify the routines for the likelihood function, the gradient vector and the Hessian. These of course depend on the problem at hand which here is the logistic distirbution.

The first routine defines the likelihood function. The code `proc likeli(b,data)` tells Gauss that the lines that follow define a subroutine called 'likeli'. It passes the arguments 'b' and 'data' to the routine and returns a single argument, a vector of the likelihoods for each observation.[7] The variables in the arguments should not be changed inside the likelihood routine. Here is a routine to calculate the likelihood in equation (2):

[7] Gauss' *maxlik* routine requires that the likelihood for each observation is calculated. Further, if analytical first derivatives are provided, then the routine that calculates them must return the score matrix.

```
proc likeli(b,data);
/*
Returns the likelihood vector for the logit model.
*/
local li,cdf,k,x,y;                 /* Define local variables */
k = cols(data);                     /* the number of indep's+1 */
x = data[.,1:k-1];                  /* the independent vars */
y = data[.,k];                      /* the dichotomous, dep.  var */
cdf = 1./(1+exp(-x*b));             /* Logit CDF */
li = y.*ln(cdf)+(1-y).*ln(1-cdf);   /* Likelihood */
retp(li);                           /* return */
endp;                               /* end of procedure */
```

The above routine is all that is needed. Given the data and the starting values GAUSS will give an answer. Indeed, as the likelihood function is globally concave, numerical derivatives work quite well here. However, providing the analytical score and the Hessian matrices, while optional, will speed up converge considerable. Especially with large data sets, the time gain can be substantial.

To calculate the score matrix use the following:

```
proc lscore(b,data);
/*
Returns the Score matrix for the logit model.
*/
local g,cdf,pdf,k,x,y;    /* The local variables */
k = cols(data);           /* the number of indep's+1 */
x = data[.,1:k-1];        /* the independent vars */
y = data[.,k];            /* the dichotomous, dep var */
cdf = 1./(1+exp(-x*b));   /* the Logit CDF */
pdf = cdf.*(1-cdf);       /* the Logit PDF */
g = y.*(pdf./cdf).*x-(1-y).*(pdf./(1-cdf)).*x;
retp(g);                  /* Return the score, g */
endp;                     /* End of porcedure */
```

In many problems, first derivatives are easy to calculate and should be, if possible, used. Indeed, the OPG estimate of the final covariance matrix of the parameters may be calculated using the routine for the score. This routine is included in the program by writing `_max_GradProc = &lscore;` before calling `maxlik`.

With large problems — thousands of observations are the rule with binary models — the inclusion of analytical second derivatives increases speed to conversion and allows one to try many model specifications in the same time that one uses for one estimation with lots of data and no derivatives. The Hessian can be calculated as below. As with the score, the routine is included in the program by writing `_max_HessProc = &lhessian;` before calling `maxlik`.

```
proc lhessian(b,data);
/*
Returns the Hessian matrix for the logit model.
*/
local pdf, k,x,y;                   /* Local variables */
k = cols(data);                     /* the number of indep's+1 */
x = data[.,1:k-1];                  /* the independent vars */
y = data[.,k];                      /* the dichotomous, dep var */
pdf = exp(-x*b)./(1+exp(-x*b))^2;   /* the Logistic PDF */
retp(-(x.*pdf)'*x);                 /* Return the Hessian */
endp;                               /* End of procedure */
```

One last item remains: we wish to see the resulats of our estimates. One way to do this is to write

`{par,f,g,H,retcode} = maxprt(maxlik(data,vars,&likli,start))`

as the statement used to start the maximization. However, one must be careful of how GAUSS calculates the covariance matrix. I usually do things by hand. The following sequence will print out the final estimates:

```
cvl = inv(-lhessian(par,data));   /* inverse of final hessian */
se = sqrt(diag(cvl));             /* standard errors */
t = par./se;                      /* t-statistics */
df = rows(data)-rows(par);        /* degreess of freedom */
sign = 2*cdftc(abs(t),df);        /* p-values */
f_lik = sumc(likli(par,data));    /* final likelihood */
/* define formats */
msk = zeros(1,1) ones(1,4);       /* one col chara data, 4 numer */
let fmt[5,3] =                    /* 5=cols of msk, 3 is required */
"*.*s " 8 8                       /* character data, 8 pos + space */
"*.*lf" 12 4                      /* left just., f-fmt, 12 pos, prec=4 */
"*.*lg" 12 4                      /* left just., f-fmt, 12 pos, prec=4 */
"*.*lg" 12 4                      /* left just., f-or e-fmt, 12 pos, prec=4 */
"*.*le" 13 3;                     /* left just., e-fmt, 13 pos, prec=3 */
print;
print "The number of iterations     :  " _max_IterData[1];
print "Final likelihood             :  " f_lik;
print "Number of observations       :  " rows(data);
print "Number of degrees of freedom :  " df
; print;
print "Variable         coef    std.err    t-stat     Signif ";
q=printfm(_max_ParNames~par~se~t~sign,msk,fmt);
```

The above rows illustrate how one can print results. Note my own 'short-hand' (with 'pos' for position, 'fmt' for format and 'just.' for justified) so the program will fit on the page.

Curt Wells
Lund, Fall 1998