Introduction to GAUSS Programming Language

Eduardo Rossi
University of Pavia

October, 2006

Contents

Chapter 1

Introduction

1.1 Introduction

GAUSS is a complex language with a large number of specialised functions for dealing with matrices. There are also a lot of add-on packages which expand GAUSS's capabilities further.

The emphasis in this course is on acquiring familiarity with the fundamentals of GAUSS and programming competence, rather than becoming a GAUSS guru.

GAUSS is a programming language designed to operate with and on matrices. It is a general purpose tool. As such, it is a long way from more specialised econometric packages. On a spectrum which runs from the computer language C at one end to, say, the menu-driven econometric program EViews at the other, GAUSS is very much at the programming end.

Advantages

1. GAUSS is appropriate for a wider range of applications than standard econometric packages because it is a general programming language.

2. GAUSS operates directly on matrices. This makes it more useful for economists than standard programming languages where the basic data units are all scalars.

3. GAUSS programs and functions are all available to the user, and so the user is able to change them.

4. Similarly, if data is held in a non-standard format, you may write your own routine to access it.

5. GAUSS is extremely powerful for matrix manipulation. It is also fast and efficient.

Disadvantages

1. The fixed costs of using GAUSS are high. Its very generality means that there is unlikely to be a simple procedure to do a simple econometric task readily to hand (although commercially available routines ameliorate this somewhat).

2. Even if pre-programmed or bought in software is available for a task, a reasonable degree of familiarity with GAUSS and its methods will often be necessary to make effective use of such routines.

3. GAUSS is too tolerant of sloppy programming. GAUSS is very flexible; however, this means it is difficult for the computer to tell when mistakes occur. For example, lax conformability requirements mean that it is easy to mistakenly divide a scalar by a row vector and then multiply by a matrix in the belief that all three variables were column vectors.

4. GAUSS is not tolerant of errors in its environment. Ask it to read from a non-existent file, or use an uninitialised variable, and the program stops. This is, of course, a sensible feature of all programming languages. Unfortunately, GAUSS is short on routines allowing non-fatal error checking.

5. Input and output routines are basic - especially input.

6. GAUSS programs are designed to be run within the GAUSS environment. They cannot be run as stand-alone programs (.EXE files) without buying a program called the GAUSS "Run-time Engine"TM. Thus you can only swap code with other GAUSS users.

1.2 Platforms and Interfaces

GAUSS is available in both single user versions and networked versions.

GAUSS on Unix is very powerful and very quick, partly because Unix machines are designed for heavy-duty processing and computation rather than user interaction. For manipulating large matrices, the time saving can be tremendous.

GAUSS on Unix runs in both teletype (command-line) and X-Windows mode. Access to the latter depends on how you access your Unix machine.

There is also a version to run on Linux (a form of Unix which runs on Intel processors).

1.2.1 Memory management

In the early days of GAUSS, efficient memory management was often crucial to getting a program running well. The amount of memory used by GAUSS could be varied by the user to make appropriate use of scarce resources. However, this is much less of an issue in modern computers, and from version 4.0 for Windows, GAUSS no longer gives you the option to manage memory directly. Instead it relies on the more efficient memory-management facilities of the operating system.

1.2.2 Interfaces

GAUSS programs can be written in two ways:

- **command-line**

  In this mode, commands typed into the GAUSS interface are executed immediately. This allows for an instant response to a command, but the commands cannot be stored. This is therefore not suitable for writing large programs, or for commands which need to be run repeatedly.

- **batch or program**

  In this mode, GAUSS commands are typed into a text file. This file is then sent to be GAUSS to be run. This allows one to develop and store complex programs.

This facility has existed since the earliest versions of GAUSS. However, the precise way this is carried out has varied over time. The original DOS interface is still extant in the latest Windows version as "TGAUSS", but the recommended interface is the windowing one. The Unix version is closer

to the DOS version but has a few operating differences. Additionally, all three versions draw graphics windows differently as a result of their operating environments.

1.3 Programming Tips

Your programs will be easier for others to follow and revise if you follow some simple tips:

1. Clean code means that do loops are indented
2. Clean code means that comments are liberally used, allowing readers to follow the program's logic.
3. A block of comments may be enclosed between two @ symbols or between /* and */. Comments delimited by the /*, */ symbols may be nested. Comments delimited by the @ symbol may not be nested. In addition, comments on a single line may be preceded by //
4. Clean code means using spaces liberally, and not necessarily including as many operations as possible in a single line of code.

1.4 Layout and Syntax

GAUSS could be described as a free-form structured language: structured because GAUSS is designed to be broken down into easily-read chunks; free-form because there is no particular layout for programs. Although the syntax is closely defined, extra spaces between words (including line breaks) are ignored. Commands are separated by a semi-colon, rather than having one command on each line as in FORTRAN or BASIC. A complete instruction is identified by the placing of semicolons, and not by the placing of commands on different lines.

Program layout is generally a matter of supreme indifference to GAUSS, and this gives the user freedom to lay out code in a style he finds acceptable.

GAUSS is not case-sensitive.

Chapter 2

The GAUSS Windows Environment

- Start GAUSS for Windows.
- You will see the *GAUSS menu bar*, the *GAUSS toolbar*, the *working directory toolbar*, a *Command Input-Output window* and, near or at the bottom of the screen, the GAUSS status bar.
- Your cursor will be in the *Command Input-Output window*, next to a GAUSS prompt, >> ,in the lower left corner of the screen.

GAUSS runs the file startup when it starts. Often users put a chdir command in the startup file to change the GAUSS working directory.

GAUSS starts up with the same configuration it had when last shutdown.

Main windows in GAUSS for Windows include:

the Command Input-Output

Edit

Output

Debug windows

Additional windows include a Matrix Editor window and an HTML Help window.

Code may be run from the:

Command Input-Output window

Edit window.

Output may appear in the:

Command Input-Output window

Output window

sent to a file.

Command Input-Output Window. Enter interactive commands and view output in the Command Input-Output window. Give this window focus by clicking inside it, by selecting it from the Windows Menu, or by typing Ctrl+W.

Edit Window. Edit windows are created and opened a number of ways.

Click the `New` button on the toolbar.

Type edit filename from the GAUSS prompt. This opens a new Edit window if filename does not already exist in the current directory. For example, typing

opens `c:\mydir\myfile.src` for editing.

It will reside in the `c:\\mydir` directory when saved.

Type `edit` *filename* from the GAUSS prompt. This opens the the editor for an existing file if filename already exists.

Click the `File/New` menu combination.

Output Window. Output is written to the Output window when it is active. GAUSS commands cannot be executed from the Output window. The Output window is made active and inactive by selecting it from the GAUSS Window menu, or by using the keyboard shortcut `Ctrl+O`. Note that `Ctrl+O` has two functions. It gives the Output window focus and changes its state (active/inactive).

Debug Window. The Debug window and Debug toolbar are displayed during a debugging session.

2.1 Startup Options

Startup options are set in the GAUSS installation directory's `startup` file or using the *Configure/Preferences* and *Configure/Editor Properties* menu items. GAUSS starts with the preferences that were in place when it was last shutdown.

Each window, including each Edit window, has its own set of options. This means that you will need to configure each window independently.

2.2 Running Commands Interactively

Single commands or blocks of commands may be run interactively from the *Command Input-Output Window*.

Select the *Command Input-Output window* by clicking inside it or by typing `Ctrl+W`.

Move to the bottom by scrolling or by simply typing `Ctrl+End`. You will see the GAUSS prompt.

Interactive commands are typically entered at the GAUSS prompt. When you run commands interactively, the processed code lies between the GAUSS prompt and the end of the current line. This is called the **active block** .

The active block can be one or more lines of code.

Pressing `Ctrl-I` will insert a GAUSS prompt in your code. This is often useful when you want to start a new active block from the middle of a line.

Enter an **active block** with more than one line of code into the *Command Input-Output window* by pressing `CTRL+Enter` at the end of each line. At the end of the final line press `Enter`.

A block of commands may be executed by selecting the code and dropping it into the *Command Input-Output window* next to the GAUSS prompt, by clicking the Run Selected Text button, or by typing `Ctrl+R`.

Repeat the last command line by pressing `CTRL+L`.

2.3 Command Keys

CTRL+A	Redo
CTRL+C	Copy selection to the Windows clipboard
CTRL+D	Open the Debug window
CTRL+E	Open the Matrix Editor
CTRL+F	Find/Replace text
CTRL+G	Go to the specified line number
CTRL+I	Insert the GAUSS prompt
CTRL+L	Insert last
CTRL+N	Make the next window active
CTRL+O	Open the Output window and change its state
CTRL+P	Print the current window, or selected text
CTRL+Q	Exit GAUSS
CTRL+R	Run selected text
CTRL+S	Save the window to a le
CTRL+W	Open the Command window
CTRL+V	Paste the contents of the Windows clipboard
CTRL+X	Cut the selection to the Windows clipboard
CTRL+Z	Undo

2.4 Function Keys

F1	Open the GAUSS Help system or context-sensitive Help
F2	Go to the next bookmark
F3	Find again
F4	Go to the next search item in Source Browser
F5	Run the Main File
F6	Run the Active File
F7	Edit the Main File
F8	Step Into
F9	Set/Clear breakpoint
F10	Step Over
ALT+F4	Exit GAUSS
ALT+F5	Debug the Main File
CTRL+F1	Searches the active libraries for the source code of a function.
CTRL+F2	Toggle bookmark
CTRL+F4	Close the active window
CTRL+F5	Compile the Main File
CTRL+F6	Compile the Active File
CTRL+F10	Step Out
ESC	Unmark marked text

2.5 Exercises

1. Type some simple GAUSS commands in the GAUSS *Command Input-Output window*, e.g.:

   ```
   let x = 1 2 3 4;
   Print x;
   ```

2. Open the *Output window*. Notice that output appears in the *Output window* when you type commands in in the *Command Input-Output window*. Switch focus to the Output window by clicking in it or by using the keystroke shortcut, `Ctrl+O`. Switch focus back and forth between the *Command Input-Output* and *Output windows*.

3. Tile the *Command Input-Output* and *Output windows*, either vertically or horizontally by selecting a Tile option from the Window menu (horizontal tiling is often useful when GAUSS output is long horizontally). Have the output from some simple commands display alternately in the *Command Input-Output* and *Output windows*.

2.6 Running Commands from Files

The GAUSS for Windows environment distinguishes between two files, the **Active file** and the **Main file**. The Active file is often called the **Current file**.

The Active file is the one displayed in the Edit window, the one with focus. An Active file may also be the Main file.

Execute the active file by clicking `Run Active File` on the `Run` menu, by clicking the `Run Currently Active File` button on the Main toolbar, or by pressing `F5`.

A file must be saved to disk before it becomes the Active file.

The Main file is displayed in the Main File list.

Execute the **Main file** by clicking `Run Main File` on the `Run` menu, by clicking the `Run Main File` button on the Main toolbar, or by pressing `F6`.

Make an **Active file** the **Main file** by choosing the `Run/Set Main File` menu item.

2.6.1 File Menu

Some useful items in the File menu are

Change Working Directory

Clear Working

Directory List

Recent Files.

Change Working Directory may also be implemented using a `chdir` or `changedir`

command or by selecting a directory from the working directory toolbar.

The ten most recent files opened are displayed at the end of the File menu. If the file you want to open is on this list, click it and GAUSS opens it in an Edit window.

2.6.2 Edit Menu

Edit window items and associated keyboard shortcuts are:

Undo	`Ctrl+Z`
Redo	`Ctrl+A`
Cut	`Ctrl+X`
Copy	`Ctrl+C`
Paste	`Ctrl+V`
Select All	
Clear All	
Find	`Ctrl+F`

The search can be case sensitive or case insensitive. You may also limit the search to regular expressions.

Find Again	`F3`
Replace	`Ctrl+Alt+F3`
Insert	`Time/Date`
Go to Line	`Ctrl+G`
Go to Next Bookmark	`F2`
Toggle Bookmark	`Ctrl+F2`
Edit Bookmarks	
Record Macro	`Ctrl+Shift+R`

2.6.3 Bookmarks

Bookmarks enable quick and easy cursor movement to particular lines or sections of code.

To add a bookmark, place the cursor in the line you want to bookmark and press `Ctrl+F2` or click the Toggle Bookmark item on the **Edit** menu.

Cycle forward through all bookmarks by pressing `F2`. Cycle backwards through all bookmarks with `Shift+F2`.

The Edit Bookmarks window lets you add, remove, name, or jump to a particular bookmark.

This window is opened when you select Edit Bookmarks from the Edit window.

Margin display must be turned on to see bookmarks. Use the Misc tab in the *Configure/Editor Properties menu* to turn margins on and off.

2.6.4 Run Menu

The **Run** Menu is used to access run commands. Run Menu items are:

Insert GAUSS Prompt Keyboard shortcut: `Ctrl+I`

Manually adds a GAUSS prompt at the cursor position.

Insert Last Command Keyboard shortcut: `Ctrl+L`

Run Selected Text Keyboard shortcut: `Ctrl+R`

This is active only when text is selected.

Run Active File Keyboard shortcut: `F6`

Runs the active file.

Test Compile Active File Keyboard shortcut: `Ctrl+F6`

This is used to see whether the Active le will compile, without entering symbols into the symbol table.

Run Main File Keyboard shortcut: `F5`

Runs the Main file.

Test Compile Main File Keyboard shortcut: `Ctrl+F5`

This is used to see whether the Main le will compile, without entering symbols into the symbol table.

Edit Main File Keyboard shortcut: `F7`

Opens an editor window for the Main le. The le is loaded, if needed, and the Edit window becomes the active window.

Stop Program Stops a program while it is running. The program will stop only after nishing the currently active GAUSS command. This sometimes takes considerable time, especially if the active command involves optimization. In this case, closing GAUSS via the taskbar is the only other way to stop the execution of a program.

Build GCG File from Main

Set Main File

Clear Main File List

Translate Dataloop Commands This must be checked if **dataloop** is being used.

2.6.5 The Help system

GAUSS has two electronic help systems, corresponding to the manuals. The "Command Reference" is an easy way to pick up information on commands (as long as they are not deemed "obsolete"), and is organised both alphbetically and by function, which is useful. It is very terse and does not display properly, but generally seems accurate.

The "User Guide" is slightly more chatty, and has more examples. Note however that it is less reliable. First, information on old commands is left unchanged, even though the commands may operate in a different way in later versions or has been deleted altogether. Second, information on new functions is often not forthcoming in that version. The first problem does seem to be disappearing, but if something doesn't work as the manual says it should, don't assume that it's your code that's wrong.

Chapter 3

GAUSS Data Types

GAUSS has two data types:

- matrices
- strings (or string arrays).

Matrices may contain numeric and character data; GAUSS does not distinguish internally between the two. Both are stored as double precision (8 byte) data.

Numeric matrices and numeric matrix elements are printed to the screen using a `print` command.

Character matrix elements are limited in length to 8 characters. Character matrix elements are printed to the screen using a

`Print $`*`variablename`* command.

Strings are printed using a

`Print` *`variablename`* command.

Matrices obviously include vectors (row and column) and scalars as subtypes, but these are all treated the same by GAUSS. For example

```
a = b + c;
```

is valid whether `a`, `b`, and `c` are scalars, vectors, or matrices, assuming the variables are conformable. However, the results of the operation may differ depending on the variable type.

Matrices may contain **numerical data** or **character data** or both. Eight bytes are used to store each element of a matrix. Hence, each cell in

a matrix can contain up to eight text characters, or numerical data with a range of about $1.0E \pm 35$.

If you enter text of more than eight characters into the cells in a matrix, the text will be truncated.

Numerical data are stored in scientific notation to around 12 places of precision.

Strings are pieces of text of unlimited length. These are used to give information to the user. If you try to assign a string value to an element of the matrix, all but the first eight characters will be lost.

Examples of data types

(4×3) Numerical matrix

$$\begin{bmatrix} 1 & 2.2 & -3 \\ 8 & 88 & 100 \\ 3.11E-6 & -2 & 5 \\ 8.6E+29 & 1000 & 2 \end{bmatrix}$$

(2×4) Character matrix

$$\begin{bmatrix} Will & Jack & Harry & Steve \\ Lory & Dick & Mary & John \end{bmatrix}$$

(3×3) Mixed matrix

$$\begin{bmatrix} Milan & 40 & K \\ Rome & 22 & M \\ London & 1 & N \end{bmatrix}$$

Strings

```
''Hello Mum!''
''Strings are pieces of text of unlimited length''
''2.2''
''''
```

The null string "" is a valid piece of text for both strings and matrices.

Because GAUSS treats all matrix data the same, GAUSS sometimes must be told that it is dealing with character data. The `$` sign identifies text and is

used in a number of places. For example, to display the value of the variable "v1" requires

```
PRINT v1; PRINT $v1;
```

or

```
PRINT v1;
PRINT $v1;
```

depending on whether `v1` is a numerical matrix, a character matrix, or a string. Strings are identified by GAUSS and don't need the `$`. You can put one in if you like but it makes no difference to printing.

Variables need to have names to reference them. Names can be any length (except in very old versions of GAUSS where they must be eight characters or less). Acceptable names for variables can contain alphanumeric data and the underscore "_", and must not begin with a number. Reserved words may not be used; standard procedure names may be reassigned, but this is not generally a good idea. Variables names are not case-sensitive.

Acceptable variable names:

`eric Eric eric1 eric_1 _eric1 _e_r_i_c`

Unacceptable variable names:

`1eric 100 if` (reserved word) `delif` (GAUSS procedure - legal, but foolish)

Using good variable names can make a big difference to your programming. Having variables called "m" and "t" may be quick to write, but "max_value" and "total_obs" would be more meaningful, and hence easier to interpret when you come back to look at a program later when you've forgotten what it does.

Grouping variables

String arrays are, as the name suggests, a convenient way of grouping strings. They are similar to a character matrix, but the strings they contain can be of unlimited length. Thus this is a valid string array:

$$\begin{bmatrix} Mediobanca & Italy \\ JPMorgan & USA \\ Schroeder & UK \end{bmatrix}$$

Note how the data fields are more than eight characters long. One difference between a **character matrix** and a **string array** is that GAUSS treats

the former as a standard array so you can carry out any matrix operation on it, whether it makes sense or not. In contrast, a lot of operations will not be allowed on a **string array** because GAUSS 'understands' the string data type.

String arrays are therefore more flexible in storing characters. However, they have some disadvantages:

1. they only store strings, and therefore you cannot mix character and numeric data.

2. because the length of the element is variable, GAUSS will handle them less efficiently. If all your character strings are eight characters or less, then keeping them in a character matrix may be marginally quicker. Third, **string arrays take up more memory**. For example, a 32768-element character matrix takes roughly 270Kb, irrespective of the number of characters. A string matrix with an average string length of 4 characters takes 400Kb; with an average length of eight characters that rises to 560Kb, twice as much as the equivalent character matrix.

Structures allow the grouping of variables of different types. They were introduced in version 4.0. Suppose you are running repeated regressions and for each regression you want to store the following information for each array:

Scalars: TSS, ESS, RSS, s, N

Vectors: Coefficients, standard errors

String array List of variable names

By placing these into a structure, they could be passed around between procedures, simplifying the program. This could also mean lower maintenance, by minimising changes to procedure calls if the structure form changes.

3.1 Creating matrices

New matrices can be defined at any point (except inside procedures). The easiest way is to assign a value to one. There are two ways to do this - by assigning a constant value or by assigning the result of some operation.

Creating a matrix using constants: LET

The keyword `LET` creates matrices. The format for creating a matrix called `varName` is

```
LET varName = constant-list;
LET varName[r,c] = constant-list;
```

In the first case, the type of matrix created depends on how the constants were specified:

- A list of constants separated by space will create a column vector.
- If the list of constants is enclosed in braces {}, then a row vector will be produced.
- When braces are used, inserting commas in the list of constants instructs GAUSS to form a matrix, breaking the rows at the commas, then an $(r \times c)$ matrix will be created; the constants will be allocated to the matrix on a row-by-row basis.
- If curly braces are not used, then adding commas has no effect. In the first case, the actual word 'LET' is optional.

If only one constant is entered, then the whole matrix will be filled with that number.

Note the **square brackets**. This is the standard way to tell GAUSS either the dimensions of a matrix or the coordinates of a block, depending on context. The first number refers to the row, the second the column. Curly braces generally are used within GAUSS to group variables together.

Examples of LET

Command Shape of x:

`LET x = 1 2 3 4 5 6;` Column vector (6×1)

`LET x = 1,2,3,4,5,6;` Column vector (6×1)

`LET x = 1 2,3 4,5 6;` Column vector (6×1)

`LET x = {1 2 3 4 5 6};` Row vector (1×6)

`LET x = {1,2,3,4,5,6};` Column vector (6×1)

`LET x = {1 2,3 4,5 6};` Matrix (3×2)

`LET x[3,2] = 1 2 3 4 5 6;` Matrix (3×2)

`LET x[3,2] = 1,2,3,4,5,6;` Matrix (3×2)

`LET x[3, 2] = 5;` Matrix (3×2)

If we have two variables "a" and "b" then the command

LET `x = a*b`;

is illegal as "a*b" is a value and not a constant. In practice, GAUSS will interpret "a*b" as a string constant and will create a string variable containing the letters and figures "a*b".

Creating a matrix using values

The results of any operation can be placed into a matrix without an LET explicit declaration. The result of the operation

```
m1 = m2 + m3;
```

will be that the value "`m2+m3`" is contained in a variable called "`m1`". If the variable `m1` did not exist before this statement, it will have been created.

The size and type of a variable depends entirely on the last thing done with it. Suppose `m1` existed prior to the last operation. If `m2` and `m3` are both scalars, then `m1` will now be a scalar - regardless of whether it was previously a matrix, vector, scalar, or string.

Variables have no fixed size or type in GAUSS - they can be changed at will simply by assigning a different value to them. It is up to the programmer to make sure he has the correct variable for any operation, as GAUSS will rarely check.

Assigning a value is done by writing down the equation. Any correct (for GAUSS's syntax) mathematical expression is acceptable, as are strings or the results of procedures.

Examples of assigning values to a variable

The routines ZEROS and ONES create matrices of 0s and 1s. The transpose operator ' can be used as in any normal equation. Examining the impact of various assignment statements on matrices `m1`, `m2` and `m3` we get

Command	`m1`	`m2`	`m3`
`m1=ZEROS(2,3);`	(2×3)	undefined	undefined
`m2=ONES(1,3);`	(2×3)	(1×3)	
`m3=m1*m2';`	(2×3)	(1×3)	(2×1)
`m1=''Hello Mum!'';`	String	(1×3)	(2×1)
`LET m2=5 2;`	String	(2×1)	(2×1)
`m3=m3'*m2;`	String	(2×1)	(1×1)

Note that LET statements can appear anywhere constants are used. The final size of `m3` will be governed by the result of the last operation; in this case, it becomes a scalar.

Why use constant assignments rather than just creating matrices as a result of mathematical or other operations? The answer is that sometimes it is awkward to create matrices of appropriate shapes. It also allows for increased security, as constant assignment is finicky about what values are appropriate, and will trap more errors.

However, you cannot rely on this. The above example of LET x = a*b giving a string variable rather than a numeric variable is a simple of how GAUSS will do the correct thing, by its definition, and happily produce a meaningless result.

In practice the main place you will use constant assignment will be at the beginning of programs where you set initial values and environment variables (like the name of an output file, or font to use for graphing). During the program you will be using variable assignment most of the time and you can ignore the strict rules on constants assignment. However, this is one of those areas where unnoticed errors creep in, and you need to be aware that GAUSS assigns values in different ways depending upon the context.

3.2 Referencing matrices

3.2.1 Direct references

Referencing strings is easy. They are one unit, indivisible. Matrices, on the other hand, are composed of the individual cells, and access to these might be required. GAUSS provides ways of accessing cells, columns, rows and blocks of the matrix as well as referring to the whole thing.

The general format is

```
mat[r1:r2,c1:c2]
```

where `mat` is the matrix and `r1, r2, c1`, and `c2` may be constants, values, or other variables. This will reference a block from row r1 to row r2, and from column c1 to column c2 of the matrix mat. A value could be assigned to this block; or this block could be extracted for output or transfer to some other location. For example, `mat = {1 2 3, 4 5 6, 7 8 9, 10 11 12};`

```
PRINT mat[2:3,1:2];
```

would print the columns 1 to 2 of rows 2 to 3 of the matrix `mat`:

$$\begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix}$$

To reference only one row or one column, only one coordinate is needed in that dimension:

`mat[r1,c1:c2]` or `mat[r1:r2,c1]`

For example, to reference the cell in the third row and fourth column of the matrix mat, these terms are all equivalent:

```
mat[3:3,4:4] mat[3,4:4] mat[3:3,4] mat[3,4]
```

Entering "." or 0 as a co-ordinate instructs GAUSS to take the whole row or column of the matrix. For example

```
mat[r1:r2,.]
```

means "rows r1 to r2 and all columns of matrix mat", while

```
mat[0, c1:c2]
```

references for columns c1 to c2.

A whole matrix could then be referred to identically as

```
mat or mat[.,.]
```

This particular feature of GAUSS causes a number of unexpected problems, particularly when using loops to access columns or rows in sequence. If your counter drops to zero (or some unspecified values) then you will find the program operating on all rows or columns instead of just one.

For vectors only one co-ordinate is needed. For a column vector, say, these are all identical

```
mat[r1:r2,.]  mat[r1:r2,0] mat[r1:r2,1] mat[r1:r2]
```

For scalars there is obviously no need for co-ordinates. However, because a scalar is a subclass of matrix,

```
mat[1,1] mat[.,.]  mat[1] mat[1,0]
```

or a number of other variations are acceptable.

This similarity in accessing matrices of zero, one, or two dimensions allows you to program loops to access matrices without necessarily knowing the dimensionality of the matrix in advance.

A last way to identify a set of rows or columns is to list them sequentially. For example, to refer to columns 1, 3, and 22 and rows 2 to 4 inclusive of the matrix mat we could use

```
mat[2:4,1 3 22]
```

Note that that there are no separating commas in the list of columns; GAUSS treats everything up to the comma as a row reference, everything afterwards as a column reference. If it finds two or more commas within square brackets, it treats this as an error.

These different methods can be combined:

```
mat[1 3:5 9, .]
```

will select every column on rows 1, 3 to 5, and 9.
The order is also important:

```
mat[1 2 3, .]
mat[3 2 1, .]
```

will give two matrices with the row order reversed in the second one.

3.2.2 Indirect references

Elements of matrices can also be referred to indirectly. Instead of explicitly using a constant to indicate a row or column number, a variable can also be used. For example,

```
PRINT mat[1:5, .];
```

```
endRow = 5;
PRINT mat[1:endRow, .];
```

are equivalent. This is a key feature in all but the most simple programs, as it avoids having to write out references explicitly. For example, suppose the program is to print out ten lines of a matrix. One solution would be to write a command to print each line:

```
PRINT mat[1,.];
PRINT mat[2,.];
...
```

This is clearly a tedious process. But one could write a loop to change the value of a variable i from 1 to 10. Then, only one PRINT statement is need in the loop:

`PRINT mat[i,.];` Even more usefully, this feature will work even if you are unsure how many lines there are in the matrix. You can set the loop to go as many times round as there are lines in the matrix. The PRINT statement does not have to be changed at all. Similarly, instead of entering explicilty a list of column or row numbers to be selected, if you enter a vector then GAUSS will use these as the indexes. For example, if `rowv` is a vector containing (1, 2, 3) then

`mat[1 2 3, .];` and `mat[rowv,.];`

are equivalent.

Nested references

Indirect references could be nested. If `rowv` and `colv` are a vectors of numbers, then

```
mat[rowv[1]:rowv[2], .]
```

is legal. So is

```
mat[rowv[r1,c1]:rowv[r2,c2], colv[rowv[r3, c3], rowv[r4,c4]]]
```

if values have been assigned to `r1, c1`... and the matrices `rowv` and `colv` have the relevant dimensions. This process can be carried on ad infinitum.

However, one problem with this flexibility in referencing is that GAUSS will always try to find a solution. For example, to access the first row of matrix mat you could use the vector `rowv` (above), one could use

```
mat[rowv[1],.]
```

However, if you omit the index

```
mat[rowv,.]
```

then GAUSS will interpret this row vector as a list of rows to be selected, as in the previous section. It will not report an error, as this construct is perfectly acceptable.

3.3 Managing data

These commands are introduced at this point as they are the basic ones for managing data. DELETE may only be used at the command line, but all the others can be included in programs.

3.3.1 SHOW

`SHOW` displays the name, size and memory location of all global variables and procedures in memory at any moment. The format is

```
SHOW varName;
SHOW/m varName;
```

where `varName` is the variable of interest. The "wild card" symbol "*" can be used, so that

```
SHOW er* ;
```

will find all references beginning with "er". The `/m` parameter means that only matrices are displayed.

3.3.2 PRINT and FORMAT

PRINT displays the contents of matrices and strings. The format is

```
PRINT var1 var2 var3...  varx ;
```

which prints the list of variables. How it prints depends on the data. If the data fits on one line (all row vectors, scalars, or strings) then PRINT will display one after the other on the same line. If, however, one of the variables is a matrix or column vector, then the variable immediately following the matrix will be printed on a new line.

PRINT wraps round when it reaches the end of the line. Each PRINT command will start off on a new line. To display without going on to a new line, the PRINT statement must be ended with two semi-colons; this stops PRINT adding a carriage return to the variable list. For example, consider

```
PRINT ''Hello'';
PRINT ''Mum''; and PRINT ''Hello'';;
PRINT ''Mum''; and PRINT ''Hello'' ''Mum'';
```

These display, respectively,

```
Hello
```

```
Mum HelloMum HelloMum
```

If strings or string constants (as above) are used, PRINT will recognise that this is character data. If, however, PRINT is given a matrix name, it must be informed if this matrix is to be printed as character data. This is done by prefixing the variable name with the dollar sign $. Hence

```
a = 1;
b = 3;
c = ''Some string data'';
d = ''char.''  | ''matrix''; PRINT a b c $d;
```

prints everything correctly. Matrices composed entirely of character data are shown in the same way; however, mixed matrices needs special commands, PRINTFM and PRINTFMT, of which more later.

Once GAUSS comes across a $ sign indicating **character data**, it prints all the rest of that line as text. Thus

```
PRINT a $c b;
```

would lead to 'b' being treated as if it were text. To get round this, 'b' must be printed in a separate statement, perhaps using the double-colon:

```
PRINT a $c;;
PRINT b;
```

PRINT style is controlled by the FORMAT commands, which sets the way matrices (but not strings) are printed. There are options to print numbers and character data with varying field widths, decimal expansion, justification, spacing and punctuation. These are covered in the manual and are all similar in form to:

```
FORMAT /RD 6, 0;
```

where, in this case, we have numbers right-justified (`/RD`), separated by spaces (`/RDC` would do commas), with 6 spaces left for writing the number and 0 decimal places. If the number is too large to fit into the space, then the field will be expanded but for that number only - not the whole matrix. Strings are given as much space as they need, but no spaces are inserted between them (see the "HelloMum" example, above).

The print styles set by FORMAT operate from the time they are set until the next FORMAT command is received.

3.3.3 NEW, CLEAR, and DELETE

These three all clean up memory. They do not affect files on disk.

NEW clears all references from memory. It can be called from inside a program, but obviously this is rarely a smart move. **The exception is at the start of a program. A call to NEW will remove any junk left over from previous work, leaving all memory free for the new program.** NEW has no parameters and is called by

```
NEW;
```

Calling NEW at the start of a program ensures that the workspace is cleared of unwanted variables, and is good practice. **Calling NEW at any other point is usually disastrous and not so highly recommended.**

CLEAR sets particular variables to zero, and it can also be called by a program. It is useful for tidying up data and initialising variables:

```
CLEAR var1 var2 ...  varN ;
```

Because it sets the variable to the scalar zero, then CLEAR is identically equal to a direct assignment:

```
CLEAR x;
```

is equivalent to x = 0;

DELETE clears variables from memory, and so is a better option than CLEAR for tidying up unwanted variables. However, it cannot be called from inside a program. The delete command is like SHOW:

```
DELETE varName;
```

```
DELETE/n varName;
```

where `varName` can include the wild card character. The `/n` option stops GAUSS double-checking the deletion is wanted. The special word "ALL" can be used instead of varName; this deletes all references, and so

```
DELETE/N ALL;
```

is equivalent to `NEW`.

Chapter 4

GAUSS procedures

- The library functions in GAUSS work like library routines in other packages - a procedure is called with some parameters, something happens, and a result may be returned.
- The parameters may be constants or variables; any returned values must be placed in variables. There may be any number of input and output parameters, including none.
- The general format in

  ```
  {outVar1, ...outVarN} = ProcName (inVar1, ...  inVarN);
  ```

  The `inVar` parameters are giving information to the procedure; the `outVar` variables are collecting information from the procedure. The input parameters will be unaffected by the action of the procedure (unless, of course, they also feature in the output list). The `outVar` parameters will be affected, and so obviously constants can not be used:

  ```
  {outVar1, ''eric''} = ThisProc (inVar1, inVar2);
  ```

  is incorrect.

Note that we have **curly brackets {}** to group variables together for the purposes of collecting results, but that we have **round brackets ()** to delineate the input parameters. The former is GAUSS's usual way of grouping things together, the latter is a near-universal programming syntax. They're mixed in together just to keep you on your toes.

If there is one or no parameter, then the form can be simplified:

```
{outVar1, ...  outVarx} = ProcName (inVar); one input parameter
{outVar1, ...  outVarx} = ProcName; no input parameter
ProcName (inVar1, ...  inVarx); no returned result
outVar = ProcName (inVar1, ...inVarx); one result returned
```

For example, the procedure DELIF requires two input parameters (a matrix and a column vector), and returns one output, a matrix:

```
outMat = DELIF (inMat, colVec);
```

The procedure EIGCG requires two input parameters and two output parameters

```
{eigsReal, eigsImag} = EIGCG(matReal, matImag);
```

The procedure SORT needs four input parameters but returns no result:

```
SORT (inFile, outFile, keyName, keyType);
```

If the program is not concerned with the results from procedure then the function CALL tells GAUSS to throw away any returns. This can save time and memory in some cases. For example, the quickest way to find the determinant of a large matrix is through a Cholesky decomposition. Running the procedure CHOL sets a global variable which can be read by the procedure DETL to give the matrix's determinant. However, the actual result of the decomposition is not wanted, only a side effect. So, to find the determinant of "mat" most quickly use

```
CALL CHOL(mat);
determ = DETL;
```

As input and returned parameters are both lists, you can pass the whole list of returned parameters to a new function, along with any other parameters that are necessary. This means that you do not need to have any intermediate variables to store the results from one procedure before passing them to another, and it will make your code shorter. However, it will not necessarily make it more readable, and you can run into maintenance problems - if you change the list of parameters for one procedure you need to change it for the other as well.

For all procedures, it is the programmer's responsibility to ensure that the right sort of data is used. If a procedure is expecting a scalar as a parameter and you pass it a row vector, for example, this will not be flagged as an error when GAUSS checks the program syntax. It may or may not cause the procedure to crash but this will not be apparent until the program is running.

All GAUSS will check is that the correct number of parameters is being passed back and forth.

4.1 Matrix algebra

Algebra involving matrices translates almost directly from the page into GAUSS. At bottom, most mathematical statements can be directly transcribed, with some small changes.

4.1.1 The basic operators

GAUSS has **eight mathematical** operators and **six relational** ones.

The mathematical ones are

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`'`	Transposition
`%`	Modulo division
`!`	Factorial
`^`	Exponentiation

The relational operators are:

`==`	`EQ`	equals
`>`	`GT`	greater than
`/=`	`NE`	does not equal
`<`	`LT`	less than
`>=`	`GE`	greater than/equals
`<=`	`LE`	less than/equals

Either the symbols or the two-letter acronyms may be used.

Note the double-equals sign for equivalence. This must not be confused with the single-equals sign implying assignment. The two return very different results:

`mat = 5; mat` is assigned the value 5; the "result" of this operation is 5

`mat ==5; mat` is compared to the value 5;

the "result" of this operation is "true" if mat is equal to 5, "false" otherwise.

With respect to logical results, GAUSS standard procedures use the convention

"false" = 0 "true" /= 0

and there are four logical operators which all return true or false

`NOT var1`	true if `var1` false, and vice-versa
`var1 AND var2`	true if `var1` true and `var2` true, else false
`var1 OR var2`	true if `var1` true or `var2` true, else false
`var1 XOR var2`	true if `var1` true or `var2` true but not both, else false
`var1 EQV var2`	true if `var1` is equivalent to `var2` i.e. both true or both false

The GAUSS manuals state that procedures set variables to 1 to signify true and 0 to false, but this is not strictly necessary - nor is it adhered to, despite several functions depending upon it. Do not rely on true==1 (eg if x==1 then...). Instead, use true/=0 (eg, if x /= 0 then...) Better still, do not rely on a particular mathematical value for true or false. GAUSS is a "strict" language: if a logical expression has several elements, all the elements of the expression will be checked even if the program has enough information to return true or false. Thus using these logical statements may be less efficient then, for example, using nested IF statements. This is also different from the way some other programs operate.

Operators work in the usual way. Thus these operations on matrices `a` to `e` are, subject to conformability requirements, all valid operations:

```
a = b+c-d;
a = b'*c';
a = (b+c)*(d-e);
a = ((b+c)*(d+e))/((b-c)*(d-e));
a = (b*c)';
```

Notice from this that matrix algebra translates almost directly into GAUSS commands. This is one of GAUSS's strong points. GAUSS will check the conformability of the above operations and reject those it finds impossible to carry out.

The order of operation is complex. But essentially the order is left to right with the following rough precedence:

brackets
transposition
factorial
exponentiation
negation
multiplication and division

addition and subtraction
dot relational operators
dot logical operators
relational operators
logical operators
row and column indices

The division operator can be used like any other. When one or other variable is a scalar, then the division operation will be carried on an element-by-element basis (see below). However, when the variables are both matrices then GAUSS will compute a generalised inverse; that is, `a = b/c` is deemed to be the solution to $ca = b$ which leads to the equations

$$a = b/c \rightarrow a = c^{-1}b$$

c square, or

$$a = (c'c)^{-1}c'b$$

c non-square.

Therefore, if two matrices are divided, then it may be preferable to do the inverse explicitly rather than leave the calculation to GAUSS. Division is a common source of unnoticed errors, because GAUSS will try as hard as possible to find an appropriate inverse.

4.1.2 Concatenation

There are two concatenation operators:

~	horizontal concatenation
|	vertical concatenation

These add one matrix to the right or bottom of another. Obviously, the relevant rows and columns must match. Consider the following operations on two matrices, `a` and `b`, with `ra` and `rb` rows and `ca` and `cb` columns, and the result placed in the matrix `c`:

dimensions `a`	dimensions `b`	operation	dimensions `c`	condition
$ra \times ca$	$rb \times cb$	`c = a~b`	$ra \times (ca + cb)$	$ra = rb$
$ra \times ca$	$rb \times cb$	`c = a | b`	$(ra + rb) \times ca$	$ca = cb$

Parts of matrices may be used, and results may be assigned to matrices or to parts:

```
a = b*c;
a = b[r1:r2,c1]*c[r3, c2:c3];
a[r1, c1:c2] = b[r1,.]*c;
```

subject to, in the last case, the recipient area being of the correct size.

These operations are available on all variables, but obviously "a=b*c′" is nonsensical when b and c are strings or character matrices. However, the relational operators may be used; and there is one useful numerical operator - addition:

```
a = b $+ c;
```

This appends c to b. Note that the operator needs the string signifier "$" to inform GAUSS to do a **string concatenation** rather than a numerical addition. If you omit the $ GAUSS will carry out a normal addition.
For example,

```
b = ''hello'';
c = ''mum'';
a = b $+ '' '' $+ c;
PRINT $a;
```

will lead to "hello mum" being printed.

With character matrices, the rules for the conformability of matrices and the application of the operator are the same as for mathematical operators. Note that, in contrast to the matrix concatenation operators, the overall matrix remains the same size (strings grow) but each of the elements in the matrix will be changed. Thus if a is an $r \times c$ matrix of file names,

```
a = a $+ ''.RES'';
```

will add the extension ".RES" to all the names in the matrix (subject to the eight-character limit) but a will still be an $(r \times c)$ matrix. If any of the cells then have more than eight characters, the extra ones are cut off.

String concatenation applied to strings and string arrays will cause these to grow.

Strings and character matrices may be compared using the relational operators. The string signifier $ is not always necessary, but it makes the program more readable and may avoid unexpected results.

In the eight bytes of data used for each matrix cell, characters and numbers are stored in different ways. GAUSS uses the $ symbol to signify the byte order, but otherwise makes no distinction between characters and num-

bers. So if you mix data types, omit a $ sign or put one in where it shouldn't be, GAUSS will not complain but the result will be gibberish.

Conformability and the "dot" operators

GAUSS generally operates in an expected way. If a scalar operand is applied to a matrix, then the operation will be applied to every element of the matrix. If two matrices are involved, the usual conformability rules apply:

Operation	Dimensions of b	Dimensions of c	Dimensions of a
`a = b * c;`	scalar	4×2	4×2
`a = b * c`	3×2	4×2	illegal
`a = b * c';`	3×2	4×2	3×4
`a = b + c;`	scalar	4×2	4×2
`a = b - c;`	3×2	4×2	illegal
`a = b - c;`	3×2	3×2	3×2

and so on.

However, GAUSS allows most of the mathematical and logical operators to be prefixed by a dot:

```
a = b.>c; a = (b+c).*d'; a = b.==c;
```

This tells the machine that operations are to be carried out on an "**element by element**" basis (or ExE, as the manual so succinctly puts it). This means that the operands are essentially broken down into the smallest conformable elements and then the scalar operators are applied. How this works in practice depends on the matrices.

To give an example, suppose that `mat1` is a (5×4) matrix. Then the following results occur for multiplication:

Operation	`mat2`	Result
mat1 * mat2	scalar	5×4; mat2 times each element of mat1
mat1 .* mat2	(5×4)	5×4; mat1[i,j] * mat2[i,j] for all i, j ($mat1 \odot mat2$)
mat1 .* mat2	(5×1)	5×4; mat1[j,i] is multiplied by mat2[j] for all i,j
mat1 .* mat2	(1×4)	5×4; mat1[j,i] is multiplied by mat2[i] for all i,j
mat1 .* mat2	anything else	illegal

Similarly for the other numerical operators:

Operation	`mat2`	Result
mat1 ./ mat2	5×4	5×4; mat1[i,j] / mat2[i,j] for all i, j
mat1 .% mat2	1×4	5×4; modulus mat1[i,j] / mat2[j] for all i, j
mat1 .*. mat2	5×4	25×16; mat1[i, j] * mat2 for all i,j ($mat1 \otimes mat2$)

The dot operators do not work consistently across all operands. In particular, for addition and subtraction no dot is needed.

Example

```
new;
x = {2 5,0.9 11};
y = {.5 .2,3 4};
z = y .*. x;
v = x *~ y;
```

$$x = \begin{bmatrix} 2 & 5 \\ 0.9 & 11 \end{bmatrix}$$

$$y = \begin{bmatrix} 0.5 & 0.2 \\ 3 & 4 \end{bmatrix}$$

$$v = \begin{bmatrix} x_{11} \times y_{11} & x_{11} \times y_{12} & x_{12} \times y_{11} & x_{12} \times y_{12} \\ x_{21} \times y_{21} & x_{21} \times y_{22} & x_{22} \times y_{21} & x_{22} \times y_{22} \end{bmatrix}$$

$$v = \begin{bmatrix} 1 & 0.4 & 2.5 & 1 \\ 2.7 & 3.6 & 33 & 44 \end{bmatrix}$$

Relational operators and dot operators

For the relational operators, the results are slightly different. These operators return a scalar 0 or 1 in normal circumstances; for example, compare two conformable matrices:

```
mat1 /= mat2;
mat1 GT mat2;
```

The first returns "true" if every element of `mat1` is not equal to every corresponding element of `mat2`; the second returns "true" if every element of `mat1` is greater than every corresponding element of `mat2`. If either variable is a scalar than the result will reflect whether every element of the matrix variable is not equal to, or greater than, the scalar. These are all scalar results.

Prefixing the operator by a dot means that the element-by-element result is returned. If `mat1` and `mat2` are both $(r \times c)$ matrices, then the results of

```
mat1 ./= mat2;
mat1 .GT mat2;
```

will be a $(r \times c)$ matrix reflecting the element-by-element result of the comparison: each cell in the result will be set to "true" or "false". If either

variable is a scalar than the result will still be a $(r \times c)$ matrix, except that each cell will reflect whether the corresponding element of the matrix variable is not equal to, or greater than, the scalar.

Fuzzy operators

In complex calculations, there will always be some element of rounding. This can lead to erroneous results from the relational operators. To avoid this, fuzzy operators are available. These are procedures which carry out comparisons within tolerance limits, rather than the exact results used by the non-fuzzy operators:

Commands	Corresponding dot operators
FEQ	DOTFEQ
FNE	DOTFNE
FGT	DOTFGT
FLT	DOTFLT
FGE	DOTFGE
FLE	DOTFGE

are used, for example `FEQ`, by

```
result = FEQ (mat1, mat2);
```

This will compare `mat1` and `mat2` to see whether they are equal within the tolerance limit, returning "true" or "false". Apart from this, the fuzzy operators (and their dot equivalents) operate as the exact relational operators.

The tolerance limit is held in a variable called `_fcmptol` which can be changed at any time. The default tolerance limit is 1.0x10-15. To change the limit simply involves giving this variable a new value:

```
_fcmptol = newValue;
```

4.2 Set operations

Column vectors can be treated like sets for some purposes. GAUSS provides three standard procedures for set operation:

- `unVec = UNION (vec1, vec2, flag);`
- `intVec = INTRSECT (vec1, vec2, flag);`
- `difVec = SETDIF (vec1, vec2, flag);`

where `unVec, intVec`, and `difVec` are the results of union, intersection, and difference operations on the two column vectors `vec1` and `vec2`. The scalar flag is used to indicate whether the data is character or numeric: 1 for numeric data, 0 for character. The difference operator returns the elements of `vec1` not in `vec2`, but not the elements of `vec2` not in `vec1`.

These commands will only work on column vectors (and obviously scalars). The two vectors can be of different sizes. A related command to the set operators is

```
unVec = UNIQUE (vec, flag);
```

which returns the column vector vec with all its duplicate elements removed and the remaining elements sorted into ascending order.

4.3 Special matrix operations

GAUSS provides methods to create and manipulate a number of useful matrix forms. The commonest are covered in this section. A fuller description is to be found in the GAUSS Command Reference.

4.3.1 Some useful matrix types

Firstly, three useful matrix creating operations:

- `identMat = EYE (iSize);`
- `onesMat = ONES (onesRows, onesCols);`
- `zerosMat = ZEROS (zeroRows, zeroCols);`

These create, respectively: an identity matrix of size `iSize`; a matrix of ones of size `onesRows` by `onesCols`; and a matrix of zeroes of size `zeroRows` by `zeroCols`.

4.3.2 Special operations

A number of common mathematical operations have been coded in GAUSS. These are simple to use to use and more efficient then building them up from scratch. They are

- `invMat = INV (mat);`
- `invPDMat = INVPD (mat);`
- `momMat = MOMENT (mat, missFlag);`
- `determ = DET (mat);`
- `determ = DETL;`
- `matRank = RANK (mat);`

The first two of these invert matrices. The matrices must be square and non-singular. `INVPD` and `INV` are almost identical except that the input matrix for `INVPD` must be symmetric and positive definite, such as a moment matrix. `INV` will work on any square invertible matrix; however, if the matrix is symmetric, then `INVPD` will work almost twice as fast because it uses the symmetry to avoid calculation. Of course, if a non-symmetric matrix is given to `INVPD`, then it will produce the wrong result because it will not check for symmetry.

GAUSS determines whether a matrix is non-singular or not using another tolerance variable. However, even if it decides that a matrix is invertible, the `INV` procedure may fail due to near-singularity. This is most likely to be a problem on large matrices with a high degree of multicollinearity. The GAUSS manual suggests a simple way to test for singularity to machine precision.

The `MOMENT` function calculates the cross-product matrix from `mat`; that is, `mat'*mat`. For anything other than small matrices, `MOMENT(x, flag)` is much quicker than using $x'x$ explicitly as GAUSS uses the symmetry of the result to avoid unnecessary operations. The missFlag instructs GAUSS what

to do about missing values - whether to ignore them (`missFlag=0`) or excise them (`missFlag=1 or 2`).

`DET` and `DETL` compute the determinants of matrices. `DET` will return the determinant of mat. `DETL`, however, uses the last determinant created by one of the standard functions; for example, `INV, DET` itself, decomposition functions all create determinants along the way. `DETL` simply reads this value. Thus `DETL` can avoid repeating calculations. The obvious drawback is that it is easy to lose track of the last matrix passed to the decomposition routines, and so determinants should be read as soon as possible after the relevant decomposition function has been called. See the Command Reference for details of which procedures create the `DETL` variable.

`RANK` calculates the rank of mat.

4.3.3 Manipulating matrices

There are a number of functions which perform useful little operations on matrices. Commonly-used ones are:

- `vec = DIAG (mat);`
- `mat = DIAGRV(mat,vec);`
- `newMat = DELIF (oldMat, flagVec);`
- `newMat = SELIF (oldMat, flagVec);`
- `newMat = RESHAPE (oldMat, newRows, newCols);`
- `nRows = ROWS (mat);`
- `nCols = COLS (mat);`
- `maxVec = MAXC (mat);`
- `minVec = MINC (mat);`
- `sumVec = SUMC (mat);`

`DIAG` and `DIAGRV` abstract and insert, respectively, a column vector from or into the diagonal of a matrix.

DELIF and SELIF allow certain rows and columns to be deleted from the matrix oldMat. The column vector flagVec has the same number of rows as oldMat and contains a series of ones and zeros. DELIF will delete all the rows from the matrix for which there is a corresponding one in flagVec, while SELIF will select all those rows and throw away the rest. Therefore DELIF and SELIF will, between themselves, cover the whole matrix.

DELIF and SELIF must have only ones and zeros in flagVec for the function to work properly. This is something to consider as the vector flagVec is often created as a result of some logical operation. For example, to delete all the rows from matrix mat1 whose first two columns are negative would involve

```
flags = (mat1[.,1] .< 0) .AND (mat1[.,2] .< 0);
mat2 = DELIF (mat1, flags);
```

This particular example should work on most systems, as the logical operator AND only returns 1 or 0. But because true is really non-zero (not 1) some operations could lead to unexpected results.

DELIF and SELIF also use a lot of memory to run.

ROWS and COLS return the number of rows and columns in the matrix of interest.

MAXC, MINC, and SUMC produce information on the columns in a matrix. MAXC creates a vector with the number of elements equal to the number of columns in the matrix. The elements in the vector are the maximum numbers in the corresponding columns of the matrix. MINC does the same for minimum values, while SUMC sums all the elements in the column. However, note that all these functions return column vectors. So, to concatenate onto the bottom of a matrix the sum of elements in each column would require an additional transposition:

```
sums = SUMC(mat1);
mat1 = mat1 | sums';
```

On the other hand, because these functions work on columns, then calling the functions again on the column vectors produced by the first call allows for matrix-wide numbers to be calculated:

```
maxMat=MAXC(MAXC(mat1));
minMat=MINC(MINC(mat1));
sumMat=SUMC(SUMC(mat1));
```

will return the largest value in mat1, the smallest value, and the total sum of the elements.

- `Y = CUMSUMC(A);` Computes the cumulative sums of the columns of a matrix.

- `Z = Complex(X,Y);` Converts a pair of real matrices to a complex matrix (`X` $(N \times K)$ real matrix, the real elements of `Z`.

- `Y = IMAG(A);` Returns the imaginary part of a matrix.

- `Y = MAXINDC(A);` Returns a column vector containing the index (i.e., row number) of the maximum element in each column in a matrix.

- `Y = MININDC(A);` Returns a column vector containing the index (i.e., row number) of the smallest element in each column in a matrix.

- `Y = RESHAPE(X,R,C);`

  Input

  `X` $(N \times K)$ matrix.

  `r` scalar, new row dimension.

  `c` scalar, new column dimension.

  Output

  `Y` $(R \times C)$ matrix created from the elements of `X`.

  The first `c` elements are put into the first row of Y, the second `c` elements in the second row, and so on.

  If there are more elements in X than in Y, the remaining are discarded. If there are not enough elements in X to fill in Y it goes back to the first elements of X and starts getting additional elements from there.

4.4 Examples

```
new;
x = {2 5,0.9 11};
y = {0.5 0.2,3 4};
z = y.*.x;
v = x.^y;
x*~y;
print z;
print v;

new;
let x[2,2] = 1 0 2 0;
let y[2,2] = 1 5 0 0;
z = .NOT x;
print z;
z = x.and y;
print z;

new;
z = ( ( .NOT x) .XOR y ) .OR Y;
print z;

new;
let x[2,2] = 1 0 2 0;
let y[2,2] = 1 5 4 2;
z = x > y;
print z;
z = x <= y;
print z;
z = x == y;
print z;
z = x .== y;
print z;
z = x ./= y;
print z;
z = (x ./=y) .AND (x.==y);
print z;
```

```
new;
x = seqa(1,1,4);
y = ones(10,4).*x';
indx = {1 3 4};
z = y[.,indx];
print z;


new;
y = rndn(100,4);
ex = y[.,2] .> 0.5;
suby = selif(y,ex);
print suby;
ex2 = abs(y[.,1]) .< 0.33 .AND y[.,2] .> 0.66;
suby2 = selif(y,ex);
print suby2;
ex = y[.,1] .< 0.33;
y = delif(y,ex);

new;
x = rndn(7*100,1); /* (700 * 1) vector */
y = reshape(x,100,7);
/*reshaping in a (100 * 7) matrix */
call printfmt(x[8:14],1);
print;
call printfmt(y[2,.],1);

new;
let A[3,3] = 1 2 3 4 5 6 7 8 9 ;
B = reshape(A,2,4);
print ''reshape(A,2,4) ='';
call printfmt(B,1);
C = reshape(A,6,4);
print ''reshape(A,6,4) ='';
call printfmt(C,1);
```

- `Y = REV(A)` Inversion of rows
- `Y = TRIMR(X,A,B)` Trimming of the first A rows and the last B rows.
- `Y = LAG1(X);` Lags a matrix by one time period for time series analysis.

  Input

  `X` $(N \times K)$ matrix.

  Output

  `Y` $(N \times K)$ matrix. First observations of y are missing.

- `Y = LAGN(X,M);` Lags or leads a matrix a specified number of time periods for time series analysis.

  Input

  `X` $(N \times K)$ matrix.

  `M` number of time periods

  `Z` $(N \times K)$ matrix.

  Output

  `Y` $(N \times K)$ matrix. If `M`>0, `lagn` lags `X` back `M` time periods, so the first `M` observations of `Y` are missing. If `M`<0, `lagn` lags `X` forward `M` time periods, so the last `M` observations of `Y` are missing.

Example

$$\texttt{X} \ (10 \times 2)$$

```
Z = Lag1(X);
```

$$Z = \begin{bmatrix} . & . \\ x_{11} & x_{12} \\ \vdots & \vdots \\ x_{91} & x_{92} \end{bmatrix}$$

```
Z = Lagn(X,2);
```

$$Z = \begin{bmatrix} . & . \\ . & . \\ x_{11} & x_{12} \\ \vdots & \vdots \\ x_{81} & x_{82} \end{bmatrix}$$

Example

```
new;
T = 10;
x = seqa(1,1,T);
xt = lagn(x,-1);
xt_1 = lag1(x);
print xt;
print xt_1;
xt = trimr(xt,0,1);
xt_1 = trimr(xt_1,1,0);
print;
print xt_1;
print xt;
```

- `Y = VEC(X);` Creates a column vector by appending the columns/rows of a matrix to each other.

  Input

  `X` $(N \times K)$ matrix.

  Output

  `Y` $(N \times K) \times 1$ vector, the columns of x appended to each other.

- `Y = SEQA(S,H,N)` creates an additive sequence

$$Y_j = S + (J-1)\,H$$

- `Y = SEQM(S,H,N)` creates a multiplicative sequence

$$Y_j = SH^{J-1}$$

- `Y = RECSERAR(X,YO,A);` computes a vector of autoregressive recursive series.

  Input

  `X` $(N \times K)$ matrix.
  `YO` $(P \times K)$ matrix.
  `A` $(P \times K)$ matrix.

  Output

  `Y` $(N \times K)$ matrix containing the series.

$$Y[t,.] = X[t,.]+A[1,.]\times Y[t-1,.]+\ldots+A[p,.]\times Y[t-p,.] \qquad t = p+1,\ldots,N$$

- `Y = RECSERCP(X,Z);` computes a recursive series involving products. Can be used to compute cumulative products.

  Input

  `X` $(N \times K)$ or $(1 \times K)$ matrix.
  `Z` $(P \times K)$ or $(1 \times K)$ matrix.

  Output

  `Y` $(N \times K)$ matrix containing the series.

$$\begin{aligned} Y[1,.] &= X[1,.] + Z[1,.] \\ Y[t,.] &= Y[t-1,.] \times X[t,.] + Z[t,.] \qquad t = 2,\ldots,N \end{aligned}$$

- `Y = RECSERRC(X,Z);` computes a recursive series involving division.
  Input

  `X` $(1 \times K)$ or $(K \times 1)$ matrix.
  `Z` $(N \times K)$ matrix.

  Output

  `Y` $(N \times K)$ matrix containing the series.

$$\begin{aligned} Y[1,.] &= X \mod Z[1], \quad X = \text{trunc}(X/Z[1]) \\ Y[2,.] &= X \mod Z[2], \quad X = \text{trunc}(X/Z[2]) \\ Y[3,.] &= X \mod Z[3], \quad X = \text{trunc}(X/Z[3]) \\ \ldots &= \ldots \\ Y[N,.] &= X \mod Z[n] \end{aligned}$$

  Can be used to convert from decimal to other number systems

4.5 Mathematical Functions

- `Y = ABS(X)` absolute value or module
- `Y = COS(X)` cosine
- `Y = SIN(X)` sine
- `Y = EXP(x)` exponential
- `Y = GAMMA(X)` $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$
- `Y = Ln(X)` natural log
- `Y = SQRT(X)` square root
- `Y = TAN(X)` tangent
- `P = CHOL(A)` Cholesky's Decomposition of a symmetric, positive definite matrix $A = P'P$, P upper triangular
- {`Va,Ve`} `= EIGHV(M)` Eigenvalues and Eigenvectors of complex hermitian or real symmetric matrix
- {`Va,Ve`} `= EIGV(M)` Eigenvalues and Eigenvectors of a general matrix
- `Va = EIGH(M)` Eigenvalues of complex hermitian or real symmetric matrix
- `Va = EIG(M)` Eigenvalues of a general matrix
- `Y = GRADP(&f,X)` Jacobian of function F in `X`
- `Y = HESSP(&f,X)` Hessian of function F in `X`
- `B = INV(A)` Inverse of `A`
- `B = INVPD(A)` Inverse of a positive definite matrix
- `B = PINV(A)` Generalised Inverse (Moore-Penrose)
- `C = POLYCHAR(A)` Characteristic Polynomial of `A`
- `C = POLYROOT(A)` Roots of a polynomial

Example

Decomposition of a symmetric positive definite matrix $\mathbf{\Sigma}$ corresponds to the command `CHOL`, i.e. Cholesky Decomposition:

$$\mathbf{\Sigma} = \mathbf{U}'\mathbf{U}$$

where $\mathbf{U}$ is an upper triangular matrix.

Let

$$\mathbf{P} = \mathbf{U}'$$

$$\mathbf{\Sigma} = \mathbf{P}\mathbf{P}'$$

Suppose we want to simulate random vectors from a multivariate normal density $N(\mu, \mathbf{\Sigma})$. Given

$$\mathbf{z} \sim N(\mathbf{0}, \mathbf{I})$$

then

$$\mathbf{x} = \mu + \mathbf{P}\mathbf{z}$$

where $\mathbf{x} \sim N(\mu, \mathbf{\Sigma})$.

```
new;
rndseed 123;
let mu = 10 12 2;
let Sigma[3,3] =
2.2 0.5 0.0
0.5 1.2 -.6
0.0 -.6 1.8;
P = chol(Sigma)';
print ''P = '' P;
print ''P*P' = '' P*P';
/* simulation of a random vector u~N(mu,Sigma) */
u = mu + P*rndn(3,1);
print ''u = '' u;
u = mu + P*rndn(3,1000); /* simulation of 1000 r.v.  */
u = u';
smean = meanc(u);
scov = vcx(u);
print ''Sample mean = '' smean;
print ''Sample covariance matrix = '' scov;
cumsumc(u)./seqa(1,1,1000);
```

4.6 Statistical Functions

- `Y = CDFN(X)` $\int_{-\infty}^{x} \phi(s)\, ds$
- `Y = CDFNC(X)` $1 - \int_{-\infty}^{x} \phi(s)\, ds$
- `Y = CDFCHIC(X,n)` Computes the complement of the cdf of the chi-square distribution.

  `Y` is the integral from x to ∞ of the chi-square distribution with n degrees of freedom.
- `Y = CDFTC(X,N)` Computes the complement of the cdf of the Students t distribution.
- `C = CORRX(X)` Correlation matrix of matrix `X`
- `M = MEANC(X)` Means of columns
- `S = STDC(X)` Standard deviation of matrix `X`'s columns
- `V = VCX(X)` Variance-covariance matrix `X`
- `N = RNDN(K,L)` $(K \times L)$ Matrix of standard normal pseudo-random numbers
- `U = RNDU(K,L)` $(K \times L)$ Matrix of uniform pseudo-random numbers
- `RNDSEED seednumber`

4.7 Examples

```
new;
rndseed 5390;
p =100;
q = 10;
x = rndn(p,q);
mu = meanc(x);
st = stdc(x);
vc = vcx(x);
print ''sample mean '' mu;
print ''sample variance'' st.^2;
print ''variance-covariance matrix'';
print;
vc;

new;
rndseed 4572;
u = rndu(1000,1);
x = 10 + 2*u;
y = x.*lag1(x).*lagn(x,2).*lagn(x,3);
y = trimr(y,3,0);
/* Correlation coefficient
```

$$\rho = \frac{\frac{1}{N}\sum_{t=1}^{N}\left(x_t - \overline{x}\right)\left(y_t - \overline{y}\right)}{\left[\frac{1}{N}\sum_{t=1}^{N}\left(x_t - \overline{x}\right)^2\right]^{1/2}\left[\frac{1}{N}\sum_{t=1}^{N}\left(y_t - \overline{y}\right)^2\right]^{1/2}}$$

```
*/

x = trimr(x,3,0);
yc = y - meanc(y);
xc = x - meanc(x);
covxy = meanc(xc.* yc);
sigmax = sqrt(meanc(xc .* xc));
sigmay = sqrt(meanc(yc.*yc));
corrxy = covxy ./ (sigmax*sigmay);
print corrxy;
corrx(y~x);
```

4.8 Strings

- `^` interprets the following name as a variable
- `Y = CHRS(X)` converts a matrix of ASCII codes to character string
- `Y = FTOCV(X,Field,Prec)` character representation of numbers in $(N \times K)$ matrix. It converts a matrix containing floating point numbers into a matrix containing the decimal character representation of each element.

  Input

  `X` $(N \times K)$ matrix containing numeric data to be converted.

  `Field` scalar, minimum field width.

  `Prec` scalar, the numbers created will have `prec` places after the decimal point.

  Output

  `Y` $(N \times K)$ matrix containing the decimal character equivalent of the corresponding elements in `X` in the format defined by `field` and `prec`.

- `Y = GETF(filename,mode)` loads ASCII or Binary file into string.

  Input

  `filename` string, any valid file name.

  `mode` scalar 1 or 0 which determines if the file is to be loaded in ASCII mode (0) or binary mode (1).

  Output

  `Y` string containing the file.

- `Y = STOF(X)` converts string to floating point

  Input

  `X` $(N \times K)$ matrix containing numeric data to be converted.

  Output

  `Y` matrix, the floating point equivalents of the ASCII numbers in x.

Example

```
n=5;
print chrs(ones(n,1)*42);
```

4.9 Printing

- `PRINTFMT(X,mask);`

  Input

  `X` $(N \times K)$ matrix which is to be printed.

  `mask` scalar, 1 if `X` is numeric or 0 if `X` is character or $(1 \times K)$ vector of 1's and 0's.

  The corresponding column of x will be printed as numeric where mask = 1 and as character where mask = 0.

- `Y = FTOS(X,Fmat,Field,Prec)` converts a scalar into a string containing the decimal character representation of that number.

  Input

  `X` scalar

  `Fmat` the format string to control the conversion

  `Field` scalar or (2×1) vector, the minimum field width

  `Prec` scalar or (2×1) vector, the numbers of places following the decimal point.

  Output

  `Y` string containing the decimal character equivalent of X in format specified.

4.10 Examples

```
Let A[4,3] = 1 2 3 4 5 6 7 8 9 10 11 12 13;
n = cols(A);
m = rows(A);


print Ftos(N, ''N= %lf'',2,0);
print Ftos(M, ''M= %lf'',2,0);
d = diag(A);
print ''Diag(A) = '';
call Printfmt(d,1);


cs = cumsumc(A);
print ''cumsumc(A) = '';
call printfmt(cs,1);


max = maxc(A);


print ''printfmt(max,1);
call printfmt(max,1);
indx = maxindc(A);


call printfmt(indx,1);
```

```
new;
let x = 125.34985027;
let y = 3.21;
let z = 5;
field = 1;
prec = 0;
fmat = ''%*.*lf'';
ftos(z,fmat,field,prec);
field = 7;
prec = 2;
fmat = ''%*.*lf seconds'';
ftos(x,fmat,field,prec);
ftos(y,fmat,field,prec);

new;
let Let A[4,3] = 1 2 3 4 5 6 7 8 9 10 11 12 13;;
v = vec(A);
vr =vecr(A)    /* vecr(A) = vec(A') */
print ''vec(A) vecr(A)'';
call printfmt(v~vr,1);
B = trimr(A,0,2);
print ''trimr(A,0,2) = '';
call printfmt(B,1);
```

4.11 Missing Values

A missing value is a special floating point encoding which the numeric processor considers a **NaN** (Not a Number).

- `Y = MISS(X,V)` elements in `X` $(N \times K)$ that are equal to the corresponding elements in `V`, $(L \times M)$ matrix $E \times E$ conformable with X, will be replaced with the GAUSS missing value code.
- `Y = MISSRV(X,V)` elements in `X` $(N \times K)$ that are equal to the GAUSS missing value code will be replaced with the corresponding element of `V` $(L \times M)$ matrix $E \times E$ conformable with `X`.
- `Y = MISSEX(X,E)` converts numeric values in `X` $(N \times K)$ to the missing value code according to the values given in a logical expression (`E` is a $(N \times K)$ logical matrix).
- `M = ISMISS(X)` Returns 1 if its matrix argument ($(N \times K)$ matrix) contains any missing values, otherwise returns a 0. The `ISMISS` function will test each element of the matrix and return 1 if it encounters any missing values.
- `M = SCALMISS(X)` Tests to see if its argument is a scalar missing value. `X` $(N \times K)$ matrix.

```
let A[3,3] = 1 2 .  4 5 6 7 .  9;
scalmiss(a);
scalmiss(a[1,3]);
```

- `y = PACKR(X)` Deletes the rows of the matrix that contain any missing values. X $(N \times K)$ matrix.
- `ENABLE;` Enables the invalid operation interrupt of the numeric processor. This affects the way missing values are handled in most calculations. When ENABLE is in effect, missing values will not be allowed in most calculations. The default is to have the program crash when missing values are encountered or when any operation sets the numeric processor's invalid operation exception. If ENABLE is on these operation will cause the program to terminate with "Invalid floating point operation" error message.

- `DISABLE;` It is the opposite of ENABLE. If DISABLE is on these operations will return a NaN, and the program will continue.

Example

```
new;
msym .;
/* The command MSYM changes the character for missing values */

machineEPSILON = machepsilon;

print ''the smallest number such that 1+ eps > 1'';
print machineEPSILON;

Missingvalue = miss(0,0);
print ftos(MissingValue, ''missing value :  %*.*s'',5,1);
print ftos(MissingValue, ''missing value :  %*.*lf'',5,1);

msym ''NA'';
print ftos(MissingValue, ''missing value :  %*.*s'',5,1);

msym -9999.99;
print ftos(MissingValue, ''missing value :  %*.*s'',5,1);
print;

let A[3,3] = 1 2 .  4 5 6 7 .  9;
print ''A ='' A;
B = missrv(A,0);
print ''B = '' B;
C = miss(A,0);
print ''C = '' C;
D = miss(a,0);
print ''D = '' D;
```

4.12 Exercises

1. Generate two random vectors (10×1), compare them $E \times E$ and return a vector of boolean variables.

2. Compare two matrices (generated randomly) and print to screen a message like "true" and "false".

3. Generate a (10×10) random matrix (using rndn) extract the rows that have the first element, i.e. the first column, greater than zero.

4. Generate a (10×10) random matrix (using rndu) and extract only the elements greater than 0.5, i.e. the output is a new matrix that has zeros in place of the elements less or equal to 0.5.

5. Create a (3×3) matrix of Gaussian pseudo-random numbers. Use a relational operator to set elements of the matrix that are less than zero to zero.

6. Create a second (3×3) matrix of Gaussian pseudo-random numbers. Generate a third matrix that equals the element-by-element products of the two matrices, conditional on the products being less than zero. All other elements of the third matrix should equal zero.

7. Create two (100×100) matrices of Uniform pseudo-random numbers. Generate a third matrix with elements equal to the sum of the first two matrices, given that each of the two elements in the sum is greater than 0.5. Elements of the third matrix are zero if this condition is not met. Use `sumc` to see whether your results roughly match the expected results (Hint: First turn the third matrix into ones and zeros).

8. Create a 10×3 matrix of pseudo-random numbers. Generate a column vector of the means of the columns of the pseudo-random matrix. Don't use `meanc` for this.

9. Create a column vector containing a sequence of numbers using `seqa`. Call it s. Create a square matrix of ones with the same number of rows and columns as rows in the sequence. Call it `X`. Notice the differences between s.*`X` and `X`.*s'. The first element-by-element multiplication causes GAUSS to sweep vertically across `X`. The second causes GAUSS to sweep horizontally, down `X`.

10. Create a string with eight or less characters. Now create a string array where each element of the array is equal to your initial string. Use the ones matrix and a multiplication operator to do this, converting the multiplication result to a string array.

11. Create a string array containing VAR1,VAR2,...

12. Create a (5×5) string array where elements in rows 2 and 3 and columns 2 and 3 are set to the letter B and the remaining elements set to the letter A.

13. Create a (5×5) matrix of missing values. First create a numeric matrix. Convert the numeric matrix to one with missing value.

Chapter 5

Flow of control

Up to now all the code used in the examples and exercises has been presented in a step-by-step way:

```
instruction1;
instruction2;
instruction3;
...
```

Both the **loop** and the **conditional branch** involve changes in the flow of control of the program: the sequence of instructions that the program executes, and the order in which they are executed, is being controlled by other instructions in the program. There are two other ways in which the sequence of instructions can be altered: by the suspension (temporary or permanent) of execution; and by procedure calls.

GAUSS also provides the ability for unconditional branching (`GOTO`, `BREAK`, `CONTINUE`) and open subroutines (`GOSUB`).

5.1 Conditional branching: IF

The syntax of the full IF statement is:

```
IF condition1;
doSomething1;
ELSEIF condition2;
```

```
doSomething2;
ELSEIF condition3;
...
ELSE;
doSomething4;
ENDIF;
```

but all the `ELSEIF` and `ELSE` statements are optional. Thus the simplest IF statement is

```
IF condition1;
doSomething1;
ENDIF;
```

- Each condition has an associated set of actions (the doSomethings).
- Each condition is tested in the order in which they appear in the program; if the condition is "true", the set of actions will be carried out.
- Once the actions associated with that condition have been carried out, and no others, GAUSS will jump to the end of the conditional branch code and continue execution from there. Thus GAUSS will only execute one set of actions at most.
- If several conditions are "true", then GAUSS will act on the first true condition found and ignore the rest.
- IF none of the conditions is met, then no action is taken, unless there is an ELSE part to the statement.
- The ELSE section has no associated condition; therefore, if GAUSS reaches the ELSE statement it will always execute the ELSE section.
- To reach the ELSE, GAUSS must have found all other conditions "false". So, ELSE is a catch-all category: it is only called when no other conditions are met, but if the ELSE section is included then some action will always be taken.
- ELSE effectively provides a default option, which can be useful in some circumstances:

```
IF number > 0 ;
    numType = ''positive'';
ELSEIF number < 0;
    numType = ''negative'';
ELSE;
    numType = ''zero'';
ENDIF;
```

or

```
numType = ''zero'';
IF number > 0;
    numType = ''positive'';
ELSEIF number < 0 ;
    numType = ''negative'';
ENDIF;
```

These programs produce identical results, but each might be appropriate in particular cases (if, for example, the default operation was very complex, or there was a need for an initialised variable numType in the branches).

IF examples

The set of actions may be one instruction, a number of instructions, or even nested IF or loop statements. It could also be a null (empty) statement. For example, augmenting the above code to separate numbers greater than one in absolute terms could be achieved by

```
new;
number = rndn(1,1);
numType = ''zero'';
IF number > 0;
    numType = ''pos '';
    IF number > 1;
        numType = numType $+ ''>1'';
    ELSE;
        numType = numType $+ ''<= 1'';
    ENDIF;
ELSEIF number < 0;
```

```
    numType = ''neg '';
    IF number < -1;
        numType = numType $+ ''<-1'';
    ELSE;
        numType = numType $+ ''>= -1'';
    ENDIF;
ENDIF;
```

Alternative formulation could be

```
new;
number = rndn(1,1);
numType = ''zero'';
IF number > 1;
    numType = ''pos >1'';
ELSEIF number > 0;
    numType = ''pos <1'';
ELSEIF number < -1;
    numType = ''neg <-1'';
ELSEIF number < 0;
    numType = ''neg > -1'';
ENDIF;
```

Finally, as a null statement is still a valid action, these three (for example) are equivalent:

```
IF condit;
    doThings;
ENDIF;
```

```
IF condit;
    doThings;
ELSE;
    ;
ENDIF;
```

```
IF condit;
    doThings;
ELSE;
ENDIF;
```

5.2 Examples

```
new;
i = 1;
do while i<=10;
number = rndn(1,1);
    IF number > 0;
        numtype = ''positive'';
    ELSEIF number <0;
        numtype = ''negative'';
    ELSE;
        numtype = ''zero'';
    ENDIF;
print numtype;
i = i + 1;
endo;

new;
i = 1;
do while i<=10;
number = rndn(1,1);
numtype = ''zero'';
IF number > 0;
    numtype = ''positive'';
    IF number > 1;
        numtype = numtype $+ '' >1'';
     ELSE;
         numtype = numtype $+ '' <= 1'';
    ENDIF;
ELSEIF number < 0;
    numtype = ''negative'';
     IF number < -1;
         numtype = numtype $+ '' < -1'';
     ELSE;
         numtype = numtype $+ '' > -1'';
     ENDIF;
ENDIF;
print numtype;
```

```
i = i + 1;
endo;

new;
x = {1 3 26, -1 3 567};
y = x < 0;
print y;
z = x.< 0;
w = x < 1000;
print w;
a = {1 2};
b = {2 0};
if a < b;
    print ''true'';
else;
    print ''false'';
endif;
if a < 2;
    print ''true'';
else;
    print ''false'';
endif;
```

```
new;
rndseed 234;
print; print ''IF1''; print;
if rndu(1,1) >= 0.5;
    print ''tail'';
endif;
print; print ''IF2''; print;
if rndu(1,1) >= 0.5;
    print ''TAIL'';
else;
    print ''HEAD'';
endif;
print; print ''IF3''; print;
i = 1;
do until i > 100;
    if rndu(1,1) >= 0.5;
else;
    print ''HEAD'';
endif;
i = i + 1;
endo;
```

5.3 Loop statements: WHILE/UNTIL and FOR

GAUSS has two types of loops: FOR loops and WHILE/UNTIL loops. The loop stops repeating itself when some condition is met. FOR loops are used when the number of loops is fixed and known in advance; the others are used when the conditions to enter or exit the loop need to be continually re-evaluated.

- WHILE/UNTIL loops. These are used when the conditions to enter or exit the loop need to be continually re-evaluated.

  These two are identical except that DO WHILE loops until condition is "false", while the DO UNTIL loops until condition is "true".

  The operation of the WHILE loop is as follows:

  1. test the condition;
  2. if "true", carry out the actions in the loop; then return to stage (1) and repeat;
  3. if "false", skip the loop actions and continue execution from the first instruction after the loop.

  The condition is tested before the loop is entered; therefore the loop might not be entered at all. Second, there is nothing in the definition of the loop to say how the loop condition is set or altered. It is the programmer's responsibility to ensure that the condition is set properly at each stage.

- FOR loops are used when the number of loops is fixed and known in advance.

5.3.1 WHILE/UNTIL loops

The format for the WHILE and UNTIL loop statements are

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

DO UNTIL condition;
doSomething;
ENDO;

These two are identical except that the first loops until condition is "false", while the second loops until condition is "true". This means that
DO WHILE condition; DO UNTIL (NOT condition);
are identical.
All the code can be converted into UNTIL statements by using the above transformation.

Two functions, BREAK and CONTINUE, allow the cycle to be interrupted. BREAK exits the loop immediately. CONTINUE takes execution back to the top of the loop where the test condition is re-evaluated. Use of these is generally a bad idea. Use IF statements to ensure an orderly exit from a loop; it makes the program much more traceable.

- BREAK breaks out of a DO or FOR loop.
- CONTINUE jumps to the top of a DO or FOR loop.

WHILE examples

Consider first of all a loop to print the integers 10 down to one. The variable `i` is used as a **count variable**: `i = 10;`

```
DO WHILE i /=0;
    PRINT i;;
    i = i - 1;
ENDO;
```

Note that the condition is set before entering the loop, and it needs to be updated explicitly, as in the penultimate line. If the line "`i = i -1;`" was not included, then `i` would have stayed at 10, the condition would not have been met, and the program would have continued printing out "10" forever. Alternatively, suppose the above code had operated on a user-entered number:

```
PRINT ''Enter start number '';;
i = CON (1, 1);
DO WHILE i /=0;
```

```
    PRINT i;;
    i = i - 1;
ENDO;
```

If the user enters a negative number to start, then `i` will never equal zero. Eventually the program will crash when i gets to -5.0E305, although this could take some days and an observant programmer may suspect that something has gone wrong before then. In this case the problem is easily avoided by changing the third line to

```
DO WHILE i > 0;
```

If the user enters a negative number with this condition, then the loop will not be executed at all.

Because the condition is tested at the beginning of a loop, the place at which the condition is changed will affect the outcome. Consider a variation on the above code:

```
i = 11;
DO WHILE i /= 1;
    i = i -1;
    PRINT i;;
ENDO;
```

This will have exactly the same result, but in the second case the condition is being changed before any action takes place, which necessitates a slight variation on the loop test and the order of instructions within the loop.

5.3.2 Examples

Generate $\sum_{i=1}^{100} x_i^2$, $x_i \in N$

```
new;
sum1 = 0;
i = 1;        /* counter  */
do while i <= 100;
    sum1 = sum1 + i^2;
    i = i + 1;
endo;
print sum1;

new;
sum2 = 0;
i = 1;
do until i > 100;
    sum2 = sum2 + i^2;
    i = i + 1;
endo;

new;
rndseed 636877789;
i = 0;
do while (RNDU(1,1) >= 0.5);
    print ''HEAD'';
    i = i + 1;
endo;
ftos(i,''The number of heads is :  %lf'',5,0);
```

This code verifies if each element of a matrix is greater than those on the boundary new;

```
x = rndn(20,10);
x = x.*x;
format 7,3;
Print ''Generated Matrix'';
print x;
print;
print;
icol = 2;
format 3,3;
do while icol < cols(x);
irang = 2;
do while irang < rows(x);
    if x[irang-1:irang+1 , icol-1:icol+1] >= x[irang,icol];
    print ''The element x[ '' irang '', '' icol '' ] satisfies
    the conditions'';
    endif;
    irang = irang + 1;
endo;
icol = icol + 1;
endo;

new;
x = rndn(4,4);
r = 0;
do while r < rows(x);
    r = r + 1;
    c = 0;
    do while c < cols(x);
        c = c + 1;
        if c == r;
            x[r,c] = 1;
        elseif c>r;
            break; /* terminate the inner loop */
        else;
            x[r,c] = 0;
```

```
        endif;
    endo;
endo;
print x;

new;
x = rndn(4,4);
r = 0;
do while r < rows(x);
    r = r + 1;
    c = 0;
    do while c < cols(x);
        c = c + 1;
        if c == r;
            continue;
        endif;
            x[r,c] = 0;
    endo;
endo;
print x;

new;
rndseed 123;
s1 = 0;
s2 = 0;
i = 1;
do until i > 100;
    e = rndu(1,1);
    if e <= 0.48;
        print ''profit'';
    s1 = s1 + 1;
    elseif e <= 0.96;
        print ''loss'';
    s2 = s2 + 1;
    else;
        print; print;
        print ftos(i,''for i = %lf, we restart'',1,0);
        print;
```

```
        continue;
    endif;
    if abs(s1 - s2) > 10;
        print; print;
        print ftos(i,''the game is stopped for i = %lf,
        we restart'',1,0);
        break;
    endif;
i = i + 1;
endo;
```

5.3.3 FOR loops

A FOR loop cycles a fixed number of times. In this it differs from the WHILE loop, which checks the end condition on every cycle. Because of this, the FOR loop operates much more quickly.

The format of the FOR loop is

```
FOR i(start, stop, interval);
:
ENDFOR;
```

- The `for` statement is faster than the `do while` and `do until` statements. However, its functionality is more limited. It uses only integers as the loop index whereas anything, an integer, complex number, ratio, string, or matrix, may be used as the loop index in a `do while` or `do until` loop. The `for` loop stops only when the specified limit value is exceeded. Stopping conditions for the `do while` and `do until` loops are more general; anything may be specified. Finally, recursive procedure calls are allowed in `do` loops but not in `for` loops.
- The variables `start`, `stop`, and `interval` control the number of times the loop operates. The loop will count from `start` to `stop` in steps of `interval`. Unlike the WHILE loop, there is no need to reset the counter explicitly.
- The counter `i` can be referenced within the loop, but should not be changed. The counter is also local to the loop. That is, if not pre-existing when the loop starts it is created; it is controlled by the loop properties; and on exiting the loop (whether normally or through BREAK) the value is undefined.

FOR examples

To rewrite the above code to count down to zero using a FOR loop, then

Example

```
FOR i (10, 1, -1);
PRINT i;;
ENDFOR;
```

This is much more compact then the corresponding WHILE loop, and will operate faster too.

The loop parameters can also be variables:

```
PRINT ''Enter start number '';;
i = CON (1, 1);FOR j(i, 1, -1);
PRINT j;;
ENDFOR;
```

Note that the start condition does not have to be explicitly tested. If the user enters a number less than 1, then the loop will not be entered at all.

$\sum_{i=1}^{99} x_i$, x_i odd

```
sum3 = 0;
for i (1,99,2);
sum3 = sum3 + i;
endfor;
ftos(sum3,''The sum of the first 50 odd numbers:  %lf'',10,0);
```

5.4 Exercises

1. Extract the diagonal of a matrix (EYE(5)) using a loop.
2. Create a diagonal matrix like:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & N \end{bmatrix}$$

   using a loop.

3. Create, using a loop, the following matrix:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 2 & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ N & N-1 & \cdots & 1 \end{bmatrix}$$

4. Generate the following matrix:

$$\begin{bmatrix} 1 & -N & 1 \times N \\ 2 & -(N-1) & 2 \times (N-1) \\ \vdots & \vdots & \vdots \\ N & -1 & N \times 1 \end{bmatrix}$$

5. Extract the lower triangular (don't use the command `Lowermat`) part of a (5×5) random matrix (use `rndn`).
6. Generate a sample $(x_1, \ldots, x_n)$ from a $X = \sum_{i=1}^{2} Y_i^2$ where $Y_i \sim N(0,1)$ and $X \sim \chi_2^2$.
   (a) Calculate the sample mean $\overline{x} = \frac{1}{n} \sum x_i$.
   (b) Repeat the step 1 and 2 R times (choose $R \geq 100$).
   (c) Calculate $\overline{\overline{x}} = \frac{1}{R} \sum_{r=1}^{R} \overline{x}_r$ and $\frac{1}{R} \sum_{r=1}^{R} \left(\overline{x}_r - \overline{\overline{x}} \right)^2$.

(d) Evaluate

$$\frac{1}{R}\sum_{r=1}^{R}\overline{x}_r - E\left[\chi_2^2\right]$$

$$\frac{1}{R}\sum_{r=1}^{R}\left(\overline{x}_r - \overline{\overline{x}}\right)^2 - Var\left[\chi_2^2\right]/n$$

Theorem 1 *Sampling Distribution of the Sample Mean.*
If $(x_1, \ldots, x_n)$ are a random sample from a population with mean μ and variance σ^2, then $\overline{x}$ is a random variable with mean μ and variance σ^2/n.

7. Sample n observations from the t distribution with d degrees of freedom by sampling *d+1* values from the standard normal distribution and then computing

$$t_{ir} = \frac{z_{ir,d+1}}{\sqrt{\frac{1}{d}\sum_{j=1}^{d} z_{ir,j}^2}} \qquad i = 1, \ldots, n \quad j = 1, \ldots, R$$

where $d = 6, 8, 20$ and $n = 10, 20, 30$.

(a) Generate, for each value of n and d, *R=100* replications.

(b) Calculate for each of the 100 replications the mean and the median.

(c) The mean and the median are unbiased estimators, so compare the mean squared errors using,

$$M_{n,d} = \frac{\frac{1}{R}\sum_{r=1}^{R}\left(median_r - 0\right)^2}{\frac{1}{R}\sum_{r=1}^{R}\left(\overline{x}_r - 0\right)^2}$$

5.5 Suspending execution: PAUSE, WAIT and END

All these commands stop execution either temporarily or permanently. In addition, some key combinations may stop a program in an emergency.

Temporary suspension using commands

Three commands can lead to the temporary suspension of a program:

```
PAUSE (sec);
WAIT;
WAITC;
```

PAUSE will wait for `sec` seconds before the program continues.

`WAIT` will wait until a key has been pressed.

`WAITC` will clear the keyboard buffer before waiting for a key, so that the program will always stop long enough for, for example, a message to be read. In this, `WAITC` works much the same as the MS-DOS "pause" command.

These functions are most useful where the program is stopped while something is being checked or a message is displayed which should be read.

WAIT and WAITC cannot be used to read console input. The key read by either of these two is lost to the program. The key is only wanted for its signalling role, not for its inherent value, and GAUSS throws the key away once the signal has been received.

Terminating a program using commands

When GAUSS has finished executing all the instructions in a file, the program is finished. However, GAUSS just returns to command mode; all the parameters, environment settings and variables used by the program still exist and are accessible to either instructions on the command line or new programs. This is the main reason for calling NEW at the beginning of a program: it clears out all the rubbish from any previous work.

Having variables around is not a problem. GAUSS could run out of memory, but as the program is finished this is unlikely to be a serious problem. However, the case for file access is different. Many PCs, and GAUSS, have some sort of disk cacheing system: a small, fast bit of memory is used as an intermediary store between disk and "normal" memory to avoid excess disk accesses. If a GAUSS dataset has been used for writing, then the last set of changes may not be permanently written to disk until the file is CLOSEd. Closing a file is the only way to be sure (relatively) that updates are properly written to disk. The GAUSS manual is silent on what happens to open files when the GAUSS environment is left. Therefore, in a worst case, running a program and then leaving the GAUSS system could result in some data being lost even though the program has run "correctly".

Ideally, a program should close all files and reset all screen and output options before it terminates. However, the command

```
END;
```

will also carry out these functions. `END` tells GAUSS that the program is complete. Even if there are more instructions, the program will terminate at this point. Moreover, the housekeeping functions will ensure that there is an orderly exit from the program. Neither NEW or `END` is necessary to a program, but between them they increase the security of the program and the integrity of the GAUSS environment. If several programs are being run, they will also improve efficiency of the programs by keeping the workspace tidy.

`END` can be placed anywhere in a program. Whenever it is encountered, the program stops. However, ENDs in the middle of a program are rarely a good idea. Having multiple exit points from a program confuses the issue, usually unnecessarily.

An alternative to `END` is

```
STOP;
```

This also indicates to GAUSS that execution is finished, but none of the housekeeping tasks are carried out. This could be used where, for example, a program had to be stopped in an emergency with files left open for examination. It is of little practical use. Use `END` in preference.

In GAUSS 5.0, `END` causes what appears to be an error message to come up at the end of the program. This can be ignored.

Chapter 6

Publication Quality Graphics

GAUSS Publication Quality Graphics consists of a set of main graphing procedures and several additional procedures and global variables for customizing the output.

Four basic parts to a graphics program:

1. Header

   In order to use the graphics procedure, the pgraph library must be active. This is done in the library statement at the top of the program.

   ```
   library pgraph;
   graphset;
   ```

   Graphset resets the graphical global variable to the default state.

2. Data setup

   The data to be graphed must be in matrices.

3. Graphics format setup

   Most of graphics elements contain defaults which allow the user to generate a plot without modification. Then defaults may be overriden by the user through the use of global variables and graphics procedures.

The variables that begins with "_p" are the global control variables used by the graphics routines.

4. Calling Graphics Routines

It is the main graphics routines where all of the work for the graphics functions get done. The graphics routines takes as input the user data and global variables which have previously been set up.

6.1 Commands

XY(x,y)	XY plot
LOGX(x,y)	Graphs X vs. Y using log coordinates for the X axes
LOGY(x,y)	Graphs X vs. Y using log coordinates for the Y axes
LOGLOG(x,y)	Graphs X vs. Y using log coordinates
BAR(val,ht)	bar graph
HIST(x,y)	Computes and graphs a frequency histogram for a vector
HISTF(x,y)	To graph a histogram given a vector of frequency counts
HISTP(x,y)	Computes and graphs a percent frequency histogram of a vector
BOX(grp,y)	Graph data using the box graph percentile method
XYZ(x,y,z)	Graphs X vs. Y vs. Z using Cartesian Coord.
SURFACE(x,y,z)	to graph a 3-D surface
CONTOUR(x,y,z)	To graph a matrix of contour data
DRAW	Graphs lines, symbols, and text using the PQG global variables

SCALE(x,y)	Fix the scaling for subsequent graphs
SCALE3D(x,y,z)	Fix the scaling
XTICS(*min,max,step,minordiv*)	To set and fix scaling, axes numbering and tick marks for the X axis to make sure that major tick marks are placed in the appropriate places
YTICS	
ZTICS	
XLABEL(str)	To set a label for the X axis
YLABEL(str)	To set a label for the Y axis
ZLABEL(str)	To set a label for the Z axis
TITLE(str)	To set the title for the graph
ASCLABEL(xl,yl)	To setup character labels for the X and Y axes
FONTS	Load fonts to be used in the graph

6.2 Global variables

_paxes scalar, 2×1, or 3×1
The first element controls the X axis, the second controls the Y axis, and the third will control the Z axis.

_paxht scalar, size of axes labels in inches.
If 0, a default size will be computed. Default = 0.

_pgrid 2×1 vector to control grid
(1,1)
0 no grid
1 dotted grid
2 fine dotted grid
3 solid grid
(2,1)
0 no subdivisions
1 dotted lines at subdiv
2 tick marks only at subdiv

_plegctl scalar or 1×4 legend control variable
(1,1)
1 coordinates in plot coordinates
2 coordinates in inches
3 coordinates in pixels
(1,2) Legend text font size. $1 \leq size \leq 9$. Default 5
(1,3) x coordinate of lower left corner of legend box
(1,4) y coordinate of lower left corner of legend box

_plegstr string, legend entry text
Text for multiple curves is separated by a null byte ("\000")

_pline $M \times 9$ matrix. Each row controls one item to be drawn.
If this is zero nothing will drawn.

_pltype scalar or $K \times 1$ vector, line type for the main curves
If this is a nonzero scalar, all lines will be this type.
If scalar 0, line types will be default styles. The default is 0
1 dashed
2 dotted
3 short dashes
4 closely spaced dots
5 dots and dashes
6 solid

_pltype	scalar or $K \times 1$ vector, line type for the main curves If this is a nonzero scalar, all lines will be this type. If scalar 0, line types will be default styles. The default is 0 1 dashed 2 dotted 3 short dashes 4 closely spaced dots 5 dots and dashes 6 solid
_plwidth	scalar or $K \times 1$ vector, line thickness for main curves Default = 0 (single pixel)
_pnum	scalar, 2×1 or 3×1 vector for independent control for axes numbering Default = 1 0 No axes numbers displayed 1 Axes numbers displayed, vertically oriented on Y axis 2 Axes numbers displayed, horizontally oriented on Y axis
_pnumht	scalar, size of axes numbers in inches. If 0 (default), a size of .13 will be used
_ptitlht	scalar, the height of the title characters in inches. If this is 0, a default height of approx. 0.13" will be used.

6.3 Fonts and Special Characters

The escape codes used for graphics text are:

\000	String termination character (null byte)
[	Enter superscript mode, leave subscript mode
]	Enter subscript mode, leave superscript mode
@	Interpret next character as literal
\20n	Select font number n
\L	create a multiple line title

Four fonts are supplied with PQG: Simplex, Complex, Simgrma and Microb. Fonts are loaded by the command `FONTS`:

```
fonts(''simplex complex microb simgrma'');
```

The `FONTS` command must be called before any of the fonts may be used in text strings. One of these fonts may be selected by embedding an escape code of the form:

\201	Simplex
\202	Complex
\203	Microb
\204	Simgrma

Default font is Simplex.

Example

```
new;
library pgraph;
graphset;
an = zeros(12,10);
x = seqa(90,1,10);
i=1;
do until i > rows(x);
    y = seqa(1,1,12);
    a = 0 $+ ''19'' $+ ftocv(x[i],2,0) $+ ftocv(y,2,0);
    an[.,i] = stof(a);
i = i + 1;
endo;
datem = vec(an);
xy(datem,rndn(rows(datem),1)+5);
```

```
new;
library pgraph;
graphset;
n = 100;
x = seqa(1,1,n);
y = sin(pi*x);
fonts(''simplex complex microb simgrma'');
title(''\201This is the title using Simplex Fonts'');
xy(x,y);
title(''\202This is the title using Complex Fonts'');
xy(x,y);
title(''\203This is the title using Microb Fonts'');
xy(x,y);
```

```
new;
library pgraph; graphset;
t = seqa(0,2*pi/100,100);
x = sin(t);
_pdate = '''';
_pnum = 2;
title(''The sinus function'');
```

```
xlabel(''t'');
ylabel(''sin(t)'');
xy(t,x);



library pgraph;
graphset;
n = 100;
x = seqa(1,1,n);
y = sin(pi*x);
fonts(''simplex simgrma'');
title( ''\201f(x) = sin(\202p\201x)'' );
xy(x,y);


x = seqa(-3,.01,6/.01);
z = sqrt(1/(2*pi))*exp(-(x.^2)/2);
title( ''\201f(x) = \2021\2011/2\202p\201e[-x[2]/2]'' );
xy(x,z);


new;
library pgraph;
graphset;
t = seqa(0,2*pi/100,100);
x1 = sin(t);
x2 = cos(t);
fonts(''simplex simgrma'');
_pdate = '''';
_pnum = 2;
title(''trigonometric functions'');
xlabel(''t'');
_plegstr = ''sin(t)\000cos(t)'';
_plegctl = {2 6 4 5};
xtics(0,2*pi,pi/2,4);
ytics(-1,1,0.5,5);
labelx = ''\2010 \202p\201/2 \202p \2013\202p\201/2 \2012\202p\201'';
```

$labelx = 0\ \pi/2\ \pi\ \frac{3\pi}{2}\ 2\pi$

```
asclabel(labelx,0);
xy(t,x1~x2);
```

6.4 Windows

A window is any rectangular subsection of the screen or page. Windows may be of any size and position on the screen and may be tiled or overlapping, transparent and nontrasparent.

Tiled windows don't overlap. The screen can be divided into any number of tiled windows with

`WINDOW`(nrows,ncols,attr);

Inputs:

1. number of rows
2. number of columns
3. window type attribute (1=transparent, 0=nontransparent).

Overlapping windows are create with the MAKEWIND command:

`MAKEWIND`(hsize,vsize,hpos,vpos,attr);

Inputs:

1. horizontal size
2. vertical size
3. horizontal distance from left edge of screen in inches
4. vertical distance from the bottom edge of screen in inches
5. window attribute type (1=transparent, 0=nontransparent)

Screen dimension expressed in inches 9×6.855

`begwind`	Window initialization procedure
`endwind`	End window manipulations, display graphs
`window`	Partition screen
`makewind`	Create Window with specified size and position
`setwind`	Set to specified window number
`nextwind`	Set to next available window number
`getwind`	Get current window number
`savewind`	Save Window configuration
`loadwind`	Load Window configuration

6.5 Examples

```
new;
library pgraph;
graphset;
t = seqa(0,2*pi/100,100);
x1 = sin(t);
x2 = cos(t);
BEGWIND;
WINDOW(1,2,0);
fonts(''simplex simgrma'');
_pdate = '''';
_pnum = 2;
xtics(0,2*pi,pi/2,4);
ytics(-1,1,.5,5);
labelx = ''\2010 \202p\201/2 \202p \2013\202p\201/2 \2012\202p\201'';
asclabel(labelx,0);
_paxht = .25;
_ptitlht = .30;
_pnumht = .35;
SETWIND(1);
title(''The sinus fucntion'');
ylabel(''sin(t)'');
xlabel(''t'');
xy(t,x1);
SETWIND(2);
title(''The cosine function'');
xlabel(''T'');
ylabel(''cos(t)'');
xy(t,x2);
ENDWIND;
```

```
new;
library pgraph;
T = seqa(1,4*pi/101,101);
x = cos(t);
y = sin(t);
z = (x'^2).*y;
n = rndn(1000,1);
graphset;
 begwind;
 makewind(9/2,6.855/2,0,0,0);
```

The MAKEWIND command will create an overlapping window of size 9/2 inches horizontally by 6.855/2 inches vertically and positioned 0 inch from the left edge of the page and 0 inch from the bottom of the page. It will be non transparent.

```
 makewind(9/2,6.855-1,9/2,1,0);
 makewind(9/2,6.855/2,0,6.885/2,0);
 makewind(9/2,6.855/2,0,6.885/2,1);
 makewind(3,3,3.5,0.25,1);
_pdate = '''';

setwind(1);
fonts(''simplex simgrma'');
title(''Parametric Function''\
''\Lx = cos(\202t\201),''\
'' y = sin(\202t\201),''\
'' z = cos(\202t\201)[2] * sin(\202t\201)''\
'' \L \202t!@[\2010,4\202p@]'');
_paxes = 0;
_pframe = 0;
_psurf = {0,0};
_pzclr = {1,9,4};
_ptitlht = .25;
volume(3,2,1);
surface(x',y,z);
setwind(2);
graphset;
```

```
title(''Histogram'');
_ptitlht = .20;
_pnumht = .20;
_paxht = .20;
_pnum = 2;
call hist(n,100);
```

HIST computes and graphs a frequency histogram for a vector. The actual frequencies are plotted for each category.

{b,m,freq}=HIST(X,V)

1. X $(M \times 1)$ vector of data
2. V $(N \times 1)$ vector of breakpoints to be used to compute the frequencies, or scalar the number of categories.

```
setwind(3);
 title(''XY Plot'');
 xtics(-2,2,1,10);
 ytics(-2,2,1,10);
 xy(x,y);
setwind(4);
 xy(2*x,2*y);
setwind(5);
 _ptitlht = .5;
 _pnum = 0;
 _paxes = 0;
 xy(2*x,2*y);
endwind;
```

```
new;
x = seqa(-10,0.1,71)';
y = seqa(-10,0.1,71);
z = cos(5*sin(x) - y);
begwind;
makewind(9,6.855,0,0,0);
makewind(9/2,6.855/2,1,1,0);
setwind(1);
_pzclr = {1,2,3,4};
title(''cos(5*sin(x) - y)'');
xlabel('' X axis'');
ylabel(''Y axis'');
scale3d(miss(0,0),miss(0,0),-5/5);
surface(x,y,z);
nextwind;
graphset;
_pzclr = {1,2,3,4};
_pbox = 15;
contour(x,y,z);
endwind;
```

Chapter 7

Input and Output

GAUSS handles data on disk in a number of formats. It can read and create standard text files and older spreadsheet formats, as well as using its own format to store matrices, datasets or code samples.

GAUSS matrices

ASCII files (normal text) anything.

GAUSS datasets .DAT, .DHT (files come in pairs)

Spreadsheets

7.1 GAUSS Matrices

A .FMT file contain a GAUSS matrix. A matrix can be saved onto disk and can be retrieved at any time. This is the default option - if no extension is given to file names, GAUSS will assume it is reading or writing a matrix file.

The commands for matrix files are:

```
load varname = filename;
```

```
loadm varname = filename;
```

If the disk file has the same name as the variable. Load and loadm are synonyms.

```
save filename = varname;
```

creates a file on disk called filename.fmt which contains the matrix varname.

```
save varname,
```

saves the variable `varname` to a file called `varname.fmt`.

Example

```
filename = ''file1'';
loadm mat1 = ^filename;
filename = ''file2'';
save ^filename = mat1;
```

This code reads a matrix from `file1.fmt` and then saves it to file `file2.fmt`.

An alternative is to have the name of the file in a string variable. To tell GAUSS that the name is contained in the string, the caret (^) operator has to be used. GAUSS then looks at the current value of the variable to see which name to use, instead of taking the variable name as a constant value.

This indirect referencing is the more usual way of using file names: it allows for the program to prompt for names, rather than having them explicitly coded into the program. This is useful when the program does not know what files are to be used - for example, if a program is to be run on several sets of data.

7.2 ASCII files

7.2.1 Reading

Input can be taken from ASCII (i.e. normal alphanumeric text) files using the LOAD command described above. This is augmented by the addition of square brackets which indicate the ASCII nature of the file:

```
load varname[]=filename;
```

or

```
load varname[r,c]=filename;
```

In the first case `varname` is a $(r \times c) \times 1$ vector if `filename` contains a $(r \times c)$ matrix.

In the second case `varname` is a $(r \times c)$ matrix. The objects in the file can be text or numbers, they must be separated by spaces or commas.

```
loadm filename;
```

This will load the data in `filename` in a variable called `filename`.

If the name of the file is a string we have to use ^ to interpret the following name as a variable.

```
filename = ''file1'';
loadm x = ^filename;
filename = ''file2'';
save ^filename = mat;
```

Remark 2 *Neither* `RESHAPE` *or* `LOAD[r, c]` *will send an error message if they do not find the correct number of elements to fill the output matrix. They will always return a matrix of the desired size. This is why it is important to check the number of elements read in before reshaping them into a matrix.*

Producing ASCII output files is no different from displaying on the screen. GAUSS allows for all output to be copied and redirected to a disk file. Thus anything which appears on the screen also appears in the disk file. To produce an ASCII file therefore requires that

1. an output file is opened;
2. PRINT is used to display all the information to go into the output file
3. the output file is closed when no more output is to be sent to it.

The relevant command to begin this process is OUTPUT:

```
OUTPUT FILE = fileName ON;
OUTPUT FILE = fileName RESET;
```

Both will instruct GAUSS to send a copy of everything it displays, from that point onward, to the file `fileName`. If `fileName` does not already exist, then these two are identical; but if the file does exist, then the first form ensures that any output is appended to the existing contents of the file, while the second empties the file before GAUSS starts writing to it. If no file name is given, then GAUSS will use the default "output.out". There is no default extension for output files.

Once a file has been opened, it can be closed and opened any number of times by combining the above commands with

```
OUTPUT OFF;
```

These commands will all work on the last recorded file name given. The `FILE=fileName` bit could be included here as well if the user wishes to swap between different output files; generally, however, only one output file is used for a program, and so naming the file explicitly is superfluous.

An analogous command `SCREEN` switches screen output on and off. These two commands are independent and so screen display off and file output on is a perfectly acceptable combination.

7.2.2 Writing

To write data to an ASCII file the print and printfmt commands are used to print to the auxiliary output. The resulting files are standard ASCII files, and can be edited with any text editor.

To write to an ASCII file we have to:

1. open an output file:

   `output file = filename on`; opens the auxiliary output file and causes the results of all print statements to be sent to that file. If the file already exists, it will be opened for appending. If the file does not already exist, it will be created.

   `output file = filename reset`; It always creates a new file. If the file already exists, it will be destroyed and a new file by that name will be created. It it does not exist, it will be created.

   Because GAUSS is treating the output as something to be "displayed" (even if only to a file), it retains the concept of only having a certain number of characters on a "line". The default is eighty characters, the standard screen width. This means that sending a matrix with a large number of columns to an output file may lead to the matrix being broken up, with "overflow" columns being put on new lines. The way to avoid this is to use

   ```
   OUTWIDTH numChars;
   ```

   where numChars is the nominal line width, and can be anything from 2 to 256. If this is set to 256, then this tells GAUSS to leave out all extraneous line breaks - new lines will only start with a new row of the matrix. Note that output on the screen may still be wrapped around. This does not affect the layout of the output file - it is just the display's functionality, and nothing to do with GAUSS.

2. print the data that we want to write;

3. close the output file:

   `output off;` closes the auxiliary output and turns off the auxiliary output.

7.2.3 Spreadsheets

GAUSS 4.0 and 5.0 for Windows can import data directly from Excel spreadsheets. For multiple-page spreadsheets, you can specify both the sheet and the cell range to upload. It will try to maintain character and numeric data characteristics.

GAUSS will also export data to these third-party formats. However, it writes these data files in the earliest compatible version. For example, although it understands Excel spreadsheets up to version 7, it will save them as version 2.1 by default.

```
mat = SPREADSHEETREADM(fileName,range,sheet);
:
okay = SPREADSHEETWRITE(mat,fileName,range,sheet);
```

In version 4: If the first row contains text, GAUSS assumes that these are column headings and creates an appropriate matrix of variable names. If it only finds numeric data, it creates a vector of column names as "C1", "C2" and so on. When exporting, the situation reverses and you can supply column names.

The import command just returns a matrix and it's up to the user to break off row or column headings. When writing, particular areas of the spreadsheet can be targetted.

Warning: while the obsolete v4.0 commands still exist in v5.0, they only work correctly if the read spreadsheet has column names. GAUSS 5 assumes that column names exist and so will automatically chop off the top row. If your spreadsheet file has no column headings, you will lose a row of data.

Using the import and export functions is much more convenient than using ASCII files as intermediaries, as well as being more reliable. However, if you are running your program on something other than GAUSS 4.0+ for Windows, you will need to go back to ASCII files for data exchange.

- `mat = xlsreadm(file, range, sheet, vls);` reads from an Excel spreadsheet, into a GAUSS matrix. `vls` is a null string or 9x1 matrix, specifies the conversion of Excel empty cells and special types into GAUSS (see Remarks). A null string results in all empty cells and empty types being converted to GAUSS missing values.

- `s = xlsreadsa(file, range, sheet, vls);` reads from an Excel spreadsheet, into a GAUSS string array or string.
- `ret = xlswritem(data, file, range, sheet, vls);` writes a GAUSS matrix to an Excel spreadsheet. `ret` is a scalar, 0 if success or a Microsoft error code.
- `ret = xlswritesa(data, file, range, sheet, vls);` writes a GAUSS string or string array to an Excel spreadsheet. `ret` is a scalar, 0 if success or a Microsoft error code.

Example

```
new;
y = seqa(1,1,10);
x = rndu(10,1);
data = y~x;
filename = ''c:\\gauss60\\eser\\gauss2excl.xls'';
range = ''a1:b10'';
ret = spreadsheetwrite(data,filename,range,1);
ret;
filename = ''c:\\gauss60\\eser\\gauss2excl2.xls'';
range = ''a1:b11'';
xlsmat = SpreadsheetReadM(filename, range, 1);
printfm(xlsmat,zeros(1,2)|ones(10,2),fmt);
```

7.2.4 Format

The following commands can be used to control printing to the auxiliary output:

`Format`

Controls the format of matrices and numbers printed out with `print` and `lprint` statements.

`format /typ /fmted /mf /jnt [f,p]`

/mf	**Matrix row format**
/m0	no delimiters
/m1 or /mb1	print 1 carriage return/line feed pair before each row of a matrix with more than 1 row
/ma1	print1 carriage return/line feed pair after each row of a matrix with more than 1 row
/a1	print 1 carriage return/line feed pair after each of a matrix
/b1	print 1 carriage return/line feed pair before each of a matrix
/jnt	**Right justified**
/rd	Signed decimal number in the form [-]####.####, where #### is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision = 0 no decimal point will be printed
/re	Signed number in the form [−]#.##E ± ###, ## is one or more decimal digits depending on the precision, and ### is three decimal digits
/ro	This will give a format like **/rd** or **/re** depending on which is the most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, a decimal point will always appear.
/rz	This will give a format like **/rd** or **/re** depending on which is the most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, trailing zeros will be suppressed and a decimal point will appear only if one or more digits follow it.

Left justified

/ld Signed decimal number in the form [-]####.####,
where #### is one or more decimal digits.
The number of digits before the decimal point
depends on the magnitude of the number,
and the number of digits after the decimal point depends on the precision.
If the precision = 0 no decimal point will be printed.
If ther number is positive a space character
will replace the lleading minus sign.

Trailing Character

- **s** the number will be followed immediately by a space character. (Default)
- **c** The number will be followed immediately with a comma.
- **t** The number will be followed immediately with a teb character.
- **n** No trailing character.

f scalar expression, controls the filed width
p scalar expression, controls the precision.

For numeric matrices, p sets the number of significant digits to be printed. For string arrays, p sets the number of characters to be printed.

A format statement stays in effect until it is overridden by a new format statement.

The total width of field will be overridden if the number is too big to fit into space allotted.

`format /rds 1,0` can be used to print integers with a single between them, regardless of the magnitudes of the integers.

GAUSS default = format /mb1 /ros 16,8

format /m1 /rd 16,8

```
new;
```

```
rndseed 444;
x = rndn(3,3);
format /m1 /rd 16,8;
print x;
```

The numbers in the matrix X will be printed with a field width of 16 spaces per number, and 8 spaces beyond the decimal point.

```
print 1000*x;
format /m1 /rd 1,8;
print x;
print 1000*x;
format /m1 /rd 13,4;
print x;
print 1000*x;
format /mb1 /ros 16,8;
```

- `Outwidth n` specifies the width of the auxiliary output. The default is 80 columns. $2 \leq n \leq 256$.
- `Screen` controls output to the screen. `screen on;` The results of all print statements will be directed to the screen. `screen off;` Print statements will not be sent to the screen. Turning the screen off will speed up execution.
- `screen;` will print "Screen is on" or "Screen is off" on the console.
- `End` terminates a program.The output file will be closed and the screen will be turned back on.

```
rndseed 124;
x = rndn(10,3);
format /rd 8,2;
outwidth 132;
output file = c:\gauss\eser\datatry1.txt reset;
screen off;
print x;
output off;
end;
```

7.3 GAUSS Dataset

GAUSS datasets are created by writing data from GAUSS or by taking an ASCII file and converting through a stand-alone program called ATOG.EXE (Ascii TO Gauss). As with the datasets for other econometric packages, they consist of rows of data split into fields. GAUSS will automatically add .dat to the filenames you give, and so there is no need to include the extension.

Unlike the GAUSS matrices, reading from or writing to a GAUSS dataset is not a single, simple operation. For matrices, the whole object is being moved into memory or onto disk. By contrast, a GAUSS dataset is used in a number of stages:

1. The file must be opened,
2. it may be read; or written to;
3. when references to the file are finished, it should be closed.

All files used will be given a handle by GAUSS; this is a scalar which is GAUSS's internal reference for that file. It will be needed for all operations on that file, and so should not be altered. The handle is needed because several files can be "open" at one time. Without the file handle, a dataset cannot be accessed, and if the file handle is overwritten then the wrong file may be used. So be careful with your handles.

7.3.1 Creating new datasets

To start a new dataset for writing, it must be created. This is done by

```
CREATE handle = filename WITH colnames,columns,type;
```

`Handle` is the handle GAUSS will return if it is successful in creating filename.

`Filename` may be a constant like "file1", or it may be a string, referenced using the ^ operator.

`Colnames` is the list of names for the columns (usually a character vector); columns tells GAUSS how many columns of data there are.

`Type` is the storage precision of the data - integers (2), single precision (4), or double precision (8).

```
filename = ''file1'';
let varnames = Date ExchgeRte InterestRte;
CREATE handle1 = ^filename WITH ^varnames,3,4;
```

It prepares a datafile called **file1.dat** for writing. A header file **file1.dht** will be also created, which records that the datafile should contain four columns, named "Date" "ExchangeRate" "InterestRate" and in single precision (type = 04, Default).

Alternatively, matrices may be converted into datasets using the command

```
Success = SAVED(variable,filename,colnames);
```

`variable` the matrix to be saved
`success` is a scalar variable set to 1 if the operation worked.

7.3.2 Opening datasets

A dataset must be opened for either reading or writing or updating (both). Once a dataset has been opened for one mode it cannot be switched to another.

```
OPEN handle = filename FOR mode VARINDXI offset;
```

handle is a non-negative scalar if the operation is successful (if the command did not work, the handle is set to -1).

`mode` = `READ`,`APPEND` or `UPDATE`. (READ = default).

If `READ` is chosen updating the file is not allowed. Choosing `APPEND` means that the data can only be appended to the file. `UPDATE` allows reading and writing.

`VARINDXI`. When GAUSS opens the file, it reads the names of fields (columns) from the .DAT fileand prefixes them all with "i" (for index). These can then be used to reference the columns of the dataset symbolically instead of using column numbers explicitly.

`Offset` scalar option shifts all these indexes by a scalar and so is useful if the data is to be concatenated horizontally to another matrix or dataset. When a file is created it is automatically opened in `APPEND` mode.

7.3.3 Reading and Moving about

A GAUSS dataset is composed of rows of data, and these rows are the basic unit of manipulation. One or more rows is read at a time; data is parcelled up into rows before being written. GAUSS mantains a file pointer which mantains the current position (i.e. row number) in the file. Generally, as rows are read from or written to the file, the row pointer is moved on. If the row pointer currently points to the start of the file and ten rows are read, the row pointer now indicates that row eleven is the current row.

To find out where the pointer currently is

```
currPos = SEEKR(handle,rownum);
```

`Handle` is the handle returned by the OPEN or CREATE.

`rownum` is the rownum to which the file pointer is to be moved. If it is set to -1, then currpos is the current row number.

To read the data, the command is

```
datamat = READR(handle,numlines);
```

which reads `numlines` rows from the file referenced by handle into the data matrix `datamat`. After the read, the file pointer will have been moved on to the point to the first row after the block just read.

7.3.4 Writing

```
result = WRITER(handle,datamat);
```

It will add `datamat` into the file at the current file position. `datamat` must have the same number of columns as the data currently in the file. **Result is the number of times actually written to disk.**

7.3.5 Closing

Files should always be closed when reading or writing is finished, GAUSS will automatically do this when leaving the GAUSS environment or when it encounters the END statement.

Files are closed by the `CLOSE` command:

```
result = CLOSE(handle);
```

$$result = \begin{cases} 0 & \text{closed successfully} \\ & \text{there is no open file attached with this handle} \\ -1 & \text{otherwise} \end{cases}$$

If the close worked, then handle should be set to 0.

```
result = close(handle);
if result == 0;
    handle = 0;
else;
print ''Close failed on file number'' handle;
endif;
```

Alternative:

```
Closeall,Closeall handle1,handle2,..,handlen;
```

Chapter 8

Procedures

Procedures are short self-contained blocks of code. When they are called by the program, the chain of command within the program switches to the procedure; when the procedure has completed all its operations, control returns to the main program.

A procedure works in just the same way as code in the main program. So why bother with them? For a number of reasons, of which the main ones are:

Tidiness. An excessively large and complicated program may be difficult to read, understand, and alter. If the program is broken into separate sections with meaningful procedure names, it becomes much more manageable.

Repetitive operations. Some functions are used in many places; for example, the READR operation, or SEQA which creates ordered vectors. The choice is between explicitly programming the same operation several times, or writing a procedure and calling it several times; usually the latter wins hands down.

Security. As the way a procedure interacts with the rest of the environment can be more strictly controlled, then procedures are often easier to test and less susceptible to unexpected influences.

The main disadvantage of procedures is the associated efficiency loss and the extra memory usage. The first is due to the overhead of setting up

subroutines and variables, and GAUSS seems to manage this relatively well. The second drawback is largely due to the need to take copies of variables, and it is the programmer's responsibility to minimise this.

8.1 Global and local variables

A variable always has a certain scope: the domain in which it is visible (accessible) to parts of a program. All of the variables considered so far have been global: they are visible to all parts of the program.

Procedures allow the use of **local variables**: they can only be seen within the ambit of the procedure. Anything outside that procedure cannot read or access those variables; as far as the program outside the procedure goes, that variable does not exist.

Local variables are only visible at the level at which they were declared. Procedures may be nested: one procedure may call another. However, the local variables are only visible to those procedures in which they were called: they are not visible to procedures they call or were called by.

Because procedures cannot see the variables created by other procedures, variables with the same name can be used in any number of procedures. If, however, variable names do conflict, (a global variable has the same name as a local variable), then the local variable always takes precedence.

Local variables only exist for the life of the procedure; once the procedure is completed and control returns to the calling code, all variables local to that procedure will be deleted from memory. If the procedure is called again, the local variables will be a completely new set, not the set that was used last time the procedure was called. Obviously, local variables always start off uninitialised.

Global variables cannot be declared inside a procedure. They may be used, their size may be changed, but they may not be declared afresh. Any variable which is used in a procedure must be either declared explicitly as a local variable or be a preexisting global variable.

8.2 Writing procedures

A procedure definition consists of five parts:

1. Procedure declaration: `proc` **statement**
2. Local variable declaration: `local` **statement**. These are variables that exist only when the procedure is executing. They cannot conflict with other variables of the same name in your main program or in other procedures.
3. Body of procedure
4. Return from procedure: `retp` **statement**
5. End of procedure definition: `endp` **statement**

```
PROC (numReturns) = ProcName( inParam1, inParam2,...  inParamN);
    LOCAL locVar1;
    :
    LOCAL locVarN;
instruction1;
instruction2;
:
instructionN;
RETP(outParam1, outParam2, ...  outParamN);
ENDP;
```

```
proc sqrtinv(x);
    local y;          @ local variable declaration @
    y = sqrt(x);     @ body of procedure @
retp(y+inv(x));     @ return from procedure @
endp;                   @ end of procedure @
```

- The first element tells GAUSS that the procedure can be referred to as `ProcName`, that it will return `numReturns` variables to the bit of code which called the procedure, and that it requires a number of pieces of information from the calling code: `inParam1` to `inParamN`. GAUSS

will check `numReturns` against the number of variables actually being returned to the calling code and produce an error message if the two do not match. It will not check that the variables are the right sort of vector, matrix, etcetera. If you have just one return value, you can omit `numReturns`.

- The input parameters are variables which can be used like any other. They are copies of the variables with which the procedure was called. Therefore they can be altered in any way inside the procedure and this will have no effect on the original variables.

- Local variables are declared using the LOCAL statement. Any variables used in the procedure which are not input parameters or global variables must be declared here. Variables can be defined in two ways:

  ```
  LOCAL x;

  LOCAL y;

  LOCAL z;
  ```

  or

  ```
  LOCAL x, y, z;
  ```

  Note that there is no information about the size or type of the variable here. All this statement says is that there are variables x, y, and z which will be accessed during this procedure, and that GAUSS should add their names to the list of valid names while this procedure is running.

- LET statements are legal in a procedure, once the variables have been identified as local, global, or parameter.

- The main body of the procedure can contain exactly the same instructions as any other section of code, with the obvious exception that procedures cannot be defined within another procedure. However, a procedure can call other procedures; the only effective limit to the number of nested procedure calls is the amount of memory available.

- When the workings of the procedure are finished, the final action is to return to the calling code any output parameters. These can be of any type; GAUSS will not check. Nor will its compiler check warn if the number of returns is not equal to `numReturns` in the procedure

declaration. GAUSS will only report an error when the procedure is actually called during a program run, so a program may run for a considerable time before an error in the number of returns is discovered.

- The RETP statement is followed by a list of output parameters. These parameters can be any of the variables used.

- If there is no value to be returned, then the RETP statement can be omitted. The procedure can have several RETPs; however, this is not recommended for the same reasons that multiple END statements are a poor idea: they confuse the flow of control, and rarely lead to more efficient programs. A RETP will usually be the penultimate line of the procedure.

- The statement ENDP tells GAUSS that the definition of the procedure is finished. GAUSS then adds the procedure to its list of symbols. It does not do anything with the code, because a procedure does not, in itself, generate any executable code. A procedure only "exists" in any meaningful sense when it is called; otherwise it is just a definition. Consider a procedure which is not called during a particular run of a program. Then that procedure could have contained any code statements and it would have made no difference whatsoever to the running of the program; for all intents and purposes, that procedure was completely ignored and might as well have been just another unused variable. This is why local variables have no existence outside their procedure: accessing variables local to a procedure that was never called is equivalent to being the child of parents who never existed.

Example

Consider this simple procedure to take a column vector and fill it with ascending numbers. The start number and increment are given as parameters. This mimics the action of the standard function SEQA:

```
PROC(1) = FillVec(inVec, startNum, step);
LOCAL i;
LOCAL nRows;
nRows = ROWS (inVec);
inVec[1] = startNum;
i = 1;
DO WHILE i <= nRows;
inVec[i] = inVec[i-1] + step;
i = i + 1;
ENDO;
RETP (inVec);
ENDP;
```

This procedure could be called by, for example,

```
sequence = FillVec(ZEROS(10, 1), 10, 10);
```

which would give a (10×1) vector counting to one hundred in tens.

In this case, even though the parameters are variables within the procedure, they were created using constants. This is due to the fact that parameters are copies of the variables passed to the procedure. In the above example, GAUSS calculated the results of the ZEROS operation; created three new variables, "inVec", "startNum", and "step", which have no further connection to the original values ZEROS(..), 10, 10; and then made these new variables visible to `FillVec`, and `FillVec` only. Thus to concatenate an index vector onto an existing matrix, a program could use

```
temp = FillVec(mat[.,1], 1, 1);
mat = mat ~temp;
```

or, equivalently and without needing an extra variable,

```
mat = mat ~FillVec(mat[.,1], 1, 1);
```

The column of `mat` used as the input vector is irrelevant; it will not be altered by the procedure call.

Note that when a procedure returns a single result, it can be treated like the result of any other operation. Thus, given a vector `iVec`, a valid command could be

```
result=sqrt((FillVec(iVec,50,1).* FillVec(iVec,50,-1)).*ones(50,
1));
```

A procedure is called the same way as an intrinsic function:

```
{zed, state1} = rndKMn(3,3,-1);
```

this statement defines the input argument to the procedure.

```
zsi = sqrtinv(zed);
```

this statement calls the procedure.

Consider a procedure which, given a GAUSS dataset handle, reads a number of lines or returns an end-of-file message:

```
PROC (2) = Extract (handle, numLines);
LOCAL currRow;
LOCAL readOkay;
LOCAL data;
currRow = SEEKR (handle, -1);
IF (currRow+numLines-1) > ROWSF(handle);
readOkay = 0;
CLEAR data;
ELSE;
readOkay = 1;
data = READR (handle, numLines);
ENDIF;
RETP (readOkay, data);
ENDP;
```

Note the need to CLEAR data: if we did not assign some value to data (in this case, 0) before we returned from the procedure, then GAUSS would report an error arising from an uninitialized variable.

8.2.1 Global Variables in External Procedures

Often procedures reference global variables, variables that exist in the global symbol table. Each procedure has its own local symbol table which vanishes after the procedure returns. The global variables in the main symbol table can be seen using the `SHOW` command.

Global variables that do not already exist in the symbol table must be declared and initialized prior to being used. The compiler learns that global variables exist from external statements, usually in **.ext** files. These files tell the compiler that a variable exists and that it is external to the procedure.

For example, suppose a collection of procedures in a file called **myprocs.src** use a global variable, `_myprocs_x`. The `myprocs.ext` file contains:

```
external matrix _myprocs_x;
```

convention is to precede globals with "_". Each entry in the **.ext** file must be initialized in a corresponding **.dec** file. A **.dec** file is where values are assigned to global variables at compile time, using declare statements, . The default initialization value for matrices is zero. The default initialization value for strings is a null string. For example, the initialization statement for the `_myprocs_x` variable looks like this:

```
declare matrix _myprocs_x; @ This takes the default value of zero @
```

Suppose there are two matrices and that you want to explicity assign values to them compile time. Your **.dec** file will contain the following lines:

```
declare matrix _myprocs_x = 3;
declare matrix _myprocs_y = { 1.2, 3, -1 };
```

8.2.2 Procedures as variables

An extremely useful feature of GAUSS is the ability to pass procedures as variables to other procedures. For example,

```
PROC(1) = Sign(mat, procVar);
LOCAL procVar:  proc;
LOCAL temp;
temp = procVar(mat);
IF temp <0;
temp = ''negative'';
ELSE;
temp = ''non-negative'';
ENDIF;
RETP (temp);
ENDP;
```

This procedure takes a procedure variable called `procVar` and a matrix `mat` as parameters. We need to declare in the procedure body that `procVar` is a procedure (by the LOCAL `procVar:  proc;` statement) so that GAUSS will realise this is a procedure and not another matrix or string.

Having done that, we can then use `procVar` within the procedure as if it were a proper procedure, even though we have no idea what the procedure is. All we require is that `procVar` takes one input parameter and returns one numeric scalar.

To use this, we need to call it with a reference to the relevant function. We do this by putting an ampersand & in front of the function name.

To continue this example, we could call the above procedure thus:

```
v = someVector;
PRINT ''The sign of the largest number is '' Sign(v, &Max_mat);
PRINT ''The sign of the smallest number is '' Sign(v, &Min_mat);
```

assuming the procedures `Max_mat` and `Min_mat` have been defined as taking a vector input and producing a scalar output. So calling any one of these functions with a vector parameter satisfies the requirements of the procedure variable `procVar`.

GAUSS does not allow GAUSS reserved words (such as MINC) to be used as procedure variables, although some standard GAUSS procedures can

be used. The list of proscribed procedures can be found in the *User Guide - Reserved Words Appendix*. Anything not in there can be used as a procedure variable.

These are trivial examples, but third-party products make extensive use of procedure variables - this is how they can supply generic optimisation routines while you just supply functions and derivatives. If you plan to use these add-on packages, it is worthwhile practising using procedure variables.

8.3 Examples

`&F` is not a procedure but a scalar it allows to recover the position of `F`.

```
new;

proc F(x);
local y;
y = x^2;
retp(y);
endp;


proc G(x);
local y;
y = x^2.*sin(x);
retp(y);
endp;


proc H(x);
local y;
y = x^2;
retp(y);
endp;


points = &F|&G|&H;


proc VE(x,i);
local f;
f = points[i];
local f:proc;
retp(F(x));
endp;


z = zeros(rows(points),1);
```

```
x = 2;
i = 1;
do while i <= rows(points);
    z[i] = VE(x,i);
i = i + 1;
endo;
print z;
```

8.4 Functions and keywords

Functions are one-line procedures which return a single parameter. They are defined slightly differently:

```
FN fnName(inParam1,...  inParamN) = someCode;
```

but otherwise operate in much the same way as procedures. However, the code in a function can only be one line, and functions do not have local variables. Thus functions can be neater than procedures for defining simple repetitive tasks, but apart from that they offer no real benefits.

Keywords take a single string as input and do not return any output. They can be useful for printing messages to the screen, for example. They are called slightly differently to procedures and functions, looking more like the PRINT function. They do allow for local variables and more than one line of code, so in that sense they are more flexible than functions. However, only taking a string as input restricts their value somewhat.

```
keyword what(str);
    print ''The argument is:''  str ;
endp;
what GAUSS;
```

8.5 Exercises

1. Write a procedure to simulate a matrix of Bernoulli random variables.
2. Write a procedure to simulate a matrix of Binomial random variables.
3. Write a procedure to compute a covariance matrix from data; simulate a data set and compute its covariance matrix.
4. Write a procedure to compute a correlation matrix from a covariance matrix.

8.6 Procedures: examples

8.6.1 Ordinary least squares

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon$$
$$\mathbf{X} \quad (N \times k)$$
$$\rho(\mathbf{X}) = k$$

$$\begin{aligned} E[\epsilon|\mathbf{X}] &= 0 \\ E[\epsilon\epsilon'|\mathbf{X}] &= \sigma_\epsilon^2 I_N \end{aligned}$$

$$y_t = \beta_1 + \beta_2 x_{2t} + \ldots + \beta_k x_{kt} + \epsilon_t$$

Parameters to estimate: $(\beta', \sigma_\epsilon^2)'$

The OLS estimator of β

$$\arg\min_\beta S(\beta) = (\mathbf{Y} - \mathbf{X}\beta)'(\mathbf{Y} - \mathbf{X}\beta)$$

$$\mathbf{X}'\widehat{\epsilon} = \mathbf{0} \rightarrow \sum_{t=1}^{T} \widehat{\epsilon}_t = 0$$
$$\widehat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{Y}$$
$$\widehat{\mathbf{Y}} = \mathbf{X}\widehat{\beta}$$
$$\widehat{\epsilon} = \mathbf{Y} - \mathbf{X}\widehat{\beta}$$
$$\widehat{\sigma}_\epsilon^2 = \frac{\widehat{\epsilon}'\widehat{\epsilon}}{T-k}$$
$$Cov\left(\widehat{\beta}\right) = \widehat{\sigma}_\epsilon^2 (\mathbf{X}'\mathbf{X})^{-1}$$
$$t\left(\widehat{\beta}_i\right) = \frac{\widehat{\beta}_i}{\sqrt{\widehat{\sigma}_\epsilon^2 (\mathbf{X}'\mathbf{X})_{ii}^{-1}}}$$
$$R^2 = \frac{\sum_{i=1}^{N} (\widehat{y}_t - \overline{y})^2}{\sum_{i=1}^{N} (y_t - \overline{y})^2} = 1 - \frac{\sum_{i=1}^{N} \widehat{\epsilon}_t^2}{\sum_{i=1}^{N} (y_t - \overline{y})^2}$$

```
new;
library pgraph;
/* Data Simulation */
n = 100;
x0 = ones(n,1);
x1 = seqa(0.02,0.001,n)+.4*rndn(n,1);
x2 = sin(seqa(1,1,n)/n+rndu(n,1));
x3 = seqm(0.2,0.02,n)+.2*rndn(n,1);
t = seqa(1,1,n);
xy(t,x1~x2~x3);
let beta[4,1] = 2,0.6,-1.7,0.5;
y = (x0~x1~x2~x3)*beta + rndn(n,1);
xy(t,y);
dati = y~(x0~x1~x2~x3);
save datimqo = dati;

/* OLS Estimate */


load dati = c:\gauss\eser\datimqo;
y = dati[.,1];
x = dati[.,2:cols(dati)];
Nobs = rows(y);
k = cols(x);
betah = invpd(x'x)*x'y;
yhat = x*betah;
epsh = y - x*betah;
ssr = epsh'epsh;
tss = (y - meanc(y))'(y - meanc(y));
rss = (yhat - meanc(y))'(yhat - meanc(y));
betah;
r2 = rss/tss;
r2;
gdl = Nobs - k;
sigmah = ssr/gdl;
Mcov = (sigmah)*invpd(x'x);
stderr = sqrt(diag(Mcov));
tstud = betah./stderr;
```

```
pvalue = 2*cdftc(abs(tstud),gdl);
deps = epsh - lag1(epsh);
deps = trimr(deps,1,0);
dw = sumc(deps^2)/sumc(epsh^2);
print ftos(Nobs,''Observation Number:  %lf'',5,0);
print ftos(k,''Number of regressors:  %lf'',5,0);
print;
print ftos(gdl,''Degrees of Freedom:  %lf'',5,0);
print;
print ftos(sigmah,''Variance Estimate:  %lf'',5,0);
print;
print ftos(DW,''Durbin-Watson:  %lf'',5,0);
print;
print '' coefficient stand.  error test t marginal prob'';
print''-------------------------------------------------'';
print betah~stderr~tstud~pvalue;
print;
print ''Estimator Var-Cov Matrix'' Mcov;
```

Transform this program in a procedure.

8.6.2 Runge - Kutta Algorithm

K Differential Equations of order 1:

$$\begin{cases} \frac{dx_1}{dt} = f_1\left(t, x_1\left(t\right), \ldots, x_k\left(t\right)\right) \\ \vdots \\ \frac{dx_i}{dt} = f_i\left(t, x_1\left(t\right), \ldots, x_k\left(t\right)\right) \\ \vdots \\ \frac{dx_k}{dt} = f_k\left(t, x_1\left(t\right), \ldots, x_k\left(t\right)\right) \end{cases}$$

Vector representation:

$$\frac{d\mathbf{x}\left(t\right)}{dt} = \mathbf{f}\left(t, \mathbf{x}\left(t\right)\right)$$

$$\mathbf{x}\left(t\right) = \begin{bmatrix} x_1\left(t\right) \\ \vdots \\ x_k\left(t\right) \end{bmatrix} \qquad \mathbf{f}\left(t, \mathbf{x}\left(t\right)\right) = \begin{bmatrix} f_1\left(t, x_1\left(t\right), \ldots, x_k\left(t\right)\right) \\ \vdots \\ f_k\left(t, x_1\left(t\right), \ldots, x_k\left(t\right)\right) \end{bmatrix}$$

h discretization step. x_i numerical solution to $x\left(t_i\right)$, where $t_i = t_0 + ih$. The algorithm solves

$$x_{i+1} = x_i + \frac{h}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]$$

$$\begin{aligned} k_1 &= f\left(t_i, x_i\right) \\ k_2 &= f\left(t_i + \frac{h}{2}, x_i + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_i + \frac{h}{2}, x_i + \frac{h}{2}k_2\right) \\ k_4 &= f\left(t_i + \frac{h}{2}, x_i + \frac{h}{2}k_3\right) \end{aligned}$$

```
/*
Format
{t,x,dx} = RuKu4(&f,xstart,tstart,tend,N):
**
Purpose
```

```
   Integrate a system of ODE using the Algortithm of Runge-Kutta
of order 4
   **
   ** Input &f - pointer to a procedure that computes f(t,X)
   ** Xstart - Kx1 vector, initial value for X
   ** tstart - scalar, initial time value
   ** tend - scalar, number of points
   **
   ** Output t - Nx1 vector, time
   ** x - Nxk, numerical solution of x(t)
   ** dx - Nxk matrix, numerical solution of dx(t)/dt
   */
   proc(3) = RuKu4(&f,Xstart,tstart,tend,N);
   local f:proc;
   local h,t,h2,K,x,dx;
   local i,ti,xi,dxi;
   local k1,k2,k3,k4,k5;
   h = (tend-tstart)/(n-1);
   t = seqa(tstart,h,N);
   h2 = h/2;
   K = rows(Xstart);
   x = zeros(K,N);
   xi = Xstart;
   x[.,1] = xi;
   dx = zeros(K,N);
   dx[.,1] = miss(zeros(K,1),0);
   i = 2;
   do until i>N;
   ti = t[i-1];
   k1 = f(ti,xi);
   k2 = f(ti+h2,xi+h2*k1);
   k3 = f(ti+h2,xi+h2*k2);
   k4 = f(ti+h,xi+h*k3);
   dxi = (h/6)*(k1+2*k2+2*k3+k4);
   xi = xi + dxi;
   x[.,i] = xi;
   dx[.,i] = dxi;
   i = i + 1;
```

```
endo;
x = x';
dx = dx';
dx = dx/h;
retp(t,x,dx);
endp;
/*
**
** Example of application of Runge-Kutta4
** dx(t)/dt = sin(t)
** x(0) = 1.5 - cos(t)
**
*/
library pgraph;
proc f(t,x);
 retp(sin(t));
endp;
t = seqa(1,1,10);
{t,x,dx} = RuKu4(&f,.5,0,10,1000);
solut = 1.5 - cos(t);
graphset;
_pdate = '''';
title(''ODE solution:  dx/dt = sin(t), x(0) = .5'');
xlabel(''t'');
ylabel(''x(t)'');
xy(t,x~solut);
```

Black & Scholes Option Pricing

$$C = S_0\Phi\left(d_1\right) - ke^{-r\tau}\Phi\left(d_2\right)$$
$$d_1 = \frac{\ln\left(S_0/K\right) + r\tau}{\sigma\sqrt{\tau}} + \frac{1}{2}\sigma\sqrt{\tau}$$
$$d_2 = d_1 - \sigma\sqrt{\tau}$$

$\Phi\left(\cdot\right)$ Gaussian CDF.

```
Proc EBSCALL(S0,K,Sigma,tau,r);
local W,d1,d2,c;
w = sigma.*sqrt(tau);
d1 = (ln(S0./K) + r.*tau)./W + .5*W,
d2 = d1 - w;
c = S0.*cdfn(d1) - K.*exp(-r.*tau).*cdfn(d2);
retp(c);
endp;
```

8.6.3 Simulation of Stochastic Differential Equation. Euler - Maruyama Algorithm

SDE:

$$\begin{aligned} \mathsf{d}X(t) &= \mu(t, X(t))\,\mathsf{d}t + \sigma(t, X(t))\,\mathsf{d}W(t) \\ X(t_0) &= X_0 \end{aligned}$$

Algorithm:

$$\begin{aligned} x_{i+1} &= x_i + \mu(t_i, x_i)\,h + \sigma(t_i, x_i)\sqrt{h}u_i \\ t_i &= t_0 + ih \\ u_i &\sim N(0,1) \end{aligned}$$

x_i simulation of $X(t_i)$

8.6.4 Simulation of the price of a European Call Option

/**Euler-Maruyama Algorithm

`x0` initial condition
`mu` process parameter (scalar or vector)
`sigma` volatility parameter (scalar or vector)
`t0` initial date
`T` final date
`N` # of discrete intervals
`M` # of simulations

**/

```
proc(2) = Eu_Maru(x0,mu,sigma,t0,T,N,M);
local mu:proc, sigma:proc;
local h,t_,x,k1,u,i,ti,xi;
h = (T-t0)/N;
t_ = seqa(t0,h,N+1);
```

```
x = zeros(N+1,M);
x[1,.]  = x0.*ones(1,M);
k1 = sqrt(h);
u = rndn(N+1,M);
i = 1;
do until i > N;
 ti = t_[i];
 xi = x[i,.];
 x[i+1,.]  = xi + mu(ti,xi)*h + k1.*sigma(ti,xi).*u[i+1,.];
 i = i + 1;
endo;
retp(t_,x);
endp;

library pgraph;
s0 = 100;
k = 98;
sigma = .15;
tau = 95/365;
r = .08;
c = EBSCall(s0,K,sigma,tau,r);
proc a(t,x);
 retp(r*x);
endp;
proc B(t,x);
 retp(sigma*x);
endp;
rndseed 1234;
Ns = 300;
{t,s} = Eu_Maru(S0,&A,&B,0,tau,100,Ns);
ST = S[101,.]';
PayOff = (ST - K).*( (St - K) .> 0);
Csimul = cumsumc(PayOff) ./seqa(1,1,Ns);
graphset;
_pdate = ''''; _pnum = 2;
title(''Monte-Carlo Simulation of European Call Option price''\
 ''\LConvergence Analysis'');
_pline = 1~1~0~C~Ns~C~1~2~10;
```

```
xlabel(''Number of Simulations'');
ylabel(''Price'');
ytics(3,7,1,10);
xy(seqa(1,1,Ns),Csimul);
proc EBScall(s0,k,sigma,tau,r);
local w,d1,d2,C;
w = sigma.*sqrt(tau);
d1 = (ln(s0./k) + r.*tau)./w + .5*w;
d2 = d1 - w;
c = s0.*cdfn(d1) - k.*exp(-r.*tau).*cdfn(d2);
retp(c);
endp;
```

8.6.5 Monte Carlo Simulation to price a European Call Option

The price of the underlying is described by a Geometric Brownian Motion

$$\begin{aligned} dS(t) &= \mu S(t)\,dt + \sigma S(t)\,dW(t) \\ S(t_0) &= S_0 \end{aligned}$$

The solution is given by the Feynman-Kac Theorem

$$C(t_0) = e^{-r\tau}\widetilde{E}\left[(S(t)-K)_+ \,|\mathfrak{F}_t\right]$$

change of probability measure by Girsanov Theorem. Under the risk neutral probability measure we have

$$\begin{aligned} dS(t) &= \mu S(t)\,dt + \sigma S(t)\,d\widetilde{W}(t) \\ S(t_0) &= S_0 \end{aligned}$$

In order to calculate the solution we can use Monte Carlo simulation

$$C(t_0) \simeq e^{-r\tau}\left[\frac{1}{N}\sum_{n=1}^{N}(S_n - K)_+\right]$$

```
library pgraph;
s0 = 100;
k = 98;
sigma = .15;
tau = 95/365;
r = .08;
c = EBSCall(s0,K,sigma,tau,r);
proc a(t,x);
 retp(r*x);
endp;
proc B(t,x);
 retp(sigma*x);
endp;
rndseed 1234;
Ns = 300;
{t,s} = Eu_Maru(S0,&A,&B,0,tau,100,Ns);
ST = S[101,.]';
PayOff = (ST - K).*( (St - K) .> 0);
Csimul = cumsumc(PayOff) ./seqa(1,1,Ns);
graphset;
_pdate = '''';  _pnum = 2;
title(''Monte-Carlo Simulation of European Call Option price''\
 ''\LConvergence Study'');
_pline = 1~1~0~C~Ns~C~1~2~10;
xlabel(''Simulations number'');
ylabel(''Price'');
ytics(3,7,1,10);
xy(seqa(1,1,Ns),Csimul);
proc EBScall(s0,k,sigma,tau,r);
local w,d1,d2,C;
w = sigma.*sqrt(tau);
d1 = (ln(s0./k) + r.*tau)./w + .5*w;
d2 = d1 - w;
c = s0.*cdfn(d1) - k.*exp(-r.*tau).*cdfn(d2);
retp(c);
endp;
```

8.6.6 Simulation of Poisson Process

Let's consider a sequence of random variables strictly increasing

$$(T_n)_{n>0}$$

- with $T_0 = 0$,

$$I_{\{T_n \leq t\}} = \begin{cases} 1 & if \quad t \geq T_n(\omega) \\ 0 & if \quad t < T_n(\omega) \end{cases}$$

  We define a *counting process* as

$$N_t = \sum_{n \geq 1} I_{\{T_n \leq t\}}$$

  A counting process is a *Poisson process* if $N_t - N_s$

- is a sequence of independent random variables
- are stationary increments (i.e. the distribution of $N_t - N_s$ is the same as $N_v - N_u$ where $0 \leq s < t < \infty$ and $0 \leq u < v < \infty$ and $t - s = v - u$)

$$\begin{aligned} N &= \{N_t\}_{0 \leq t \leq \infty} \\ \Pr\{N_t = n\} &= \frac{e^{-\lambda t}(\lambda t)^n}{n!} \quad n = 0, 1, 2, \ldots \quad \lambda > 0 \end{aligned}$$

```
/* Simulation of Poisson Process */
/*
Input
lambda - scalar, Process parameter
t0 - scalar, starting date
T - scalar, final date
L - scalar, number of discrete steps
K - scalar, number of simulations
Output.
ti - vector (L+1)x1, dates
N - matrix (L+1)xK, simulated Poisson process
The i-th simulated process corresponds to the vector N[.,i]
```

```
*/

proc(2)=poisson(lambda,t0,T,L,K);
local h,ti,u,N;
    h = (T-t0)/L;
    ti = seqa(t0,h,L+1);
    u = rndu(L,k);
    u = u.< (lambda*h);
    u = zeros(1,K)|u;
    N = cumsumc(u);
retp(ti,N);
endp;

library pgraph;
rndseed 123;
L = 1000;
t0 = 0;
TT = 10;
lambda = 5;

{t,N} = poisson(lambda,t0,TT,L,1);
{t,N2} = poisson(2,t0,TT,L,1);

graphset;
fonts(''simplex simgrma'');
_pdate = '''';
_plwidth = 5;
_pnum = 2;
title(''Simulation of Poisson process''\
 ''\L\2021\201 = 5 _ _ \2021\201 =2'');
xtics(t0,TT,1,0);
ytics(0,50,10,0);
xs = t[1:L];
xe = t[2:L+1];
ys = N[1:L];
ye = N2[1:L];
e = ones(L,1);
xlabel(''t'');
```

```
   ylabel(''N]t[(\202w\201)'');
   xy(xs,ys~ye);
```

8.6.7 Simulation of Diffusion Process with jumps

```
/*
   Simulation of diffusion process with jumps
   Input
   x0 - scalar or vector Nsx1, initial value of the process
   &mu - pointer di una procesura che calcola la funzione mu(t,X)
   &sigma - pointer to a procedure that calculates the function sigma(t,X)
   &kappa - pointer to a procedure that calculates the function kappa(t,X)
   &lambda - pointer to a procedure that calculates the function
lambda(t,X)
   t0 - scalar, t0
   TT - scalar, T
   N - scalar, # of points of discretization of the interval [t0,T]
   Ns - scalare, number of simulations

   Output
   t - vector (N+1)x1, dates ti
   xt - matrix (N+1)xNs, simulated diffusion process with jumps
   dN - matrix (N+1)xNs, when jump 1, no jump 0.
   */

   new;
   proc(3) = salto(x0,mu,sigma,kappa,lambda,t0,TT,N,Ns);
   local h,t,xt,u,dN,i,xi,ti,dN_,lambda_;
   local mu:proc, sigma:proc,kappa:proc,lambda:proc;

   h = (TT - t0)/N;
   t = seqa(t0,h,N+1);
   xt = zeros(N+1,Ns);
   xt[1,.]  = x0.*ones(1,Ns);
   u = rndn(N,Ns)*sqrt(h);
   dN = zeros(N+1,Ns);
   i = 1;
   do until i > N;
```

```
xi = xt[i,.];
ti = t[i];
lambda_ = lambda(ti,xi);
dN_ = rndu(1,Ns) .< (lambda_*h);
xt[1+i,.]  = xi + mu(ti,xi)*h + sigma(ti,xi).*u[i,.] + kappa(ti,xi).*dN_;
dn[1+i,.]  = dN_;
i = i + 1;
endo;
retp(t,xt,dN);
endp;
```

$$
\begin{aligned}
dX(t) &= \mu X(t)\,dt + \sigma X(t)\,dW(t) + k(t)\,dN(t) \\
X(0) &= 2 \\
k(t) &= 1 \\
\lambda(t,X) &= \begin{cases} 1 & \text{if } t \leq 5 \\ 0.5 & \text{otherwise} \end{cases}
\end{aligned}
$$

We can obtain as a by-product the Geometric Brownian Motion just setting lambda = 0 or kappa = 0.

```
library pgraph;
x0 = 2;
proc mu(t,x);
retp(0.0002*x);
endp;
proc sigma(t,x);
retp(1.09*x);
endp;
proc kappa(t,x);
retp(1);
endp;
proc lambda1(t,x);
retp(0);
endp;
```

```
rndseed 123;
{t,x1,dN} = salto(x0,&mu,&sigma,&kappa,&lambda1,0,10,100,1);
proc lambda2(t,x);
if t <= 5;
retp(1);
else;
retp(.5);
endif;
endp;
rndseed 123;
{t,x2,dN} = salto(x0,&mu,&sigma,&kappa,&lambda2,0,10,100,1);
xs2 = indexcat(dN[.,1],1);
graphset;
fonts(''simplex simgrma'');
_pdate = '''';
_pnum = 2;
title(''Geometric Brownian Motion :  ''\
 ''dX]t[ = \202m\201X]t[dt + \202s\201X]t[dW]t[''\
 ''\LJump Process :  ''\
 ''dX]t[ = \202m\201X]t[dt + \202s\201X]t[ + \202k\201]t[dN]t['');
xy(t,x1~x2);
```

Chapter 9

Libraries

9.1 The GAUSS library system

The GAUSS library system allows for the creation and maintenance of modular programs. The user can create libraries of frequently used functions that the GAUSS system will automatically find and compile whenever they are referenced in a program.

The first place GAUSS looks for a symbol definition is in the active libraries. A GAUSS library is a text file that serves as a dictionary to the source files that contain the symbol definitions.

The command LIBRARY is used to activate libraries. The libraries are in a subdirectory listed in the configuration file with lib_path.

The library files have the extension .lcg.

To create a library

1. Create a file containing procedures, functions, keywords, etc.

```
/* NORM.SRC */
proc onenorm(x);
    retp(maxc(sumc(abs(x))));
endp;
proc infnorm(x);
    retp(maxc(sumc(abs(x'))));
endp;
```

   By convention, these files are named with an .SRC extension.

2. Create the library file (GAUSS assigns to it the default extension .LCG) with the command LIB

   ```
   LIB matem norm.src;
   ```

3. In this way in the library matem.lcg is contained the file norm.src., which in turn contains the three procedures that are in NORM.SRC.

9.1.1 Autoloader

The autoloader resolves references to procedures, keywords, matrices and strings that are not defined in the program from which they are referenced. The autoloader automatically locates and compiles the files containing the symbol definitions that are not resolved during the compilation of the main file. The search path used by the autoloader is first the current directory, and then the paths listed in the SRC_PATH configuration variable in the order they appear. SRC_PATH can be defined in the GAUSS configuration file.

9.1.2 User LIBRARY

It is the library for the current procedures. If the autoloader is on, it is the first library where GAUSS looks for to solve the reference to symbols not defined.

```
LIB USER standard.src;
```

9.1.3 File .G

If the autoloader is on and a symbol cannot be found in a library, the autoloader will assume it is a library and look for a file that has the same name as the symbol with an extension .G in one of the subdirectories listed in the SRC_PATH.

9.1.4 Global Variables

If the application makes use of several global variables, it is better to create a file containing DECLARE statements. We use files with the extension

.DEC to assign default values to global matrices and strings with declare statements.

9.1.5 External Variables

A file with an .EXT extension containing the same symbols in external statements can also be create and #INCLUDE'd at the top of any file that references these global variables. An appropriate library file should contain the name of the .DEC files and the names of the globals they declare.

9.2 Example: Portfolio Optimization

File MARKO.EXT

```
/* File with the declarations of external matrices */
external matrix _Markowitz_A;
external matrix _Markowitz_B;
external matrix _Markowitz_C;
external matrix _Markowitz_D;
external matrix _Markowitz_bnds;
```

File MARKO.DEC

```
/* File with the declarations of global matrices */
declare matrix _Markowitz_A = 0;
declare matrix _Markowitz_B = 0;
declare matrix _Markowitz_C = 0;
declare matrix _Markowitz_D = 0;
declare matrix _Markowitz_bnds = 0;
```

File MARKO.SRC

/**

The Markowitz mean-variance determination of optimal portfolio

Problem

$$\max E(w) - \frac{\phi}{2}\sigma(w)$$

$$\begin{aligned} E(w) &= \alpha'\mu \\ \sigma(w) &= \alpha' V \alpha \\ s.t. \quad \iota'\alpha &= 1 \\ \alpha_i &\geq 0 \end{aligned}$$

```
{alpha,Rp,Sigmap} = markowitz(mu,Mcov,Phi);
```

Input
`Phi` scalar, preference parameter (risk aversion)
`mu` vector (Nx1), expected returns
`Mcov` matrix (NxN), returns variance-covariance matrix V

Output
`Rp` scalar, optimal portfolio mean return
`alpha` vector (Nx1), optimal portfolio weights
`Sigmap` scalar, optimal portfolio risk
**/

```
#include marko.ext;


proc(3) = markowitz(mu,Mcov,phi);
local numass;
local sv,Q,R,A,B,C,D,bnds;
local alpha,u1,u2,u3,u4,retcode;
local portreturn, portrisk;
numass = rows(mu);
sv = ones(numass,1)/numass;
Q = phi*Mcov;
R = mu;
if _Markowitz_A == 0;
 A = ones(1,numass);
else;
 A = _Markowitz_A;
endif;
if _markowitz_B == 0;
 B = 1;
else;
```

```
 B = _markowitz_B;
endif;
if _Markowitz_C == 0;
 C = eye(numass);
else;
 C = _markowitz_C;
endif;
if _Markowitz_D == 0;
 D = zeros(numass,1);
else;
 D = _markowitz_D;
endif;
if _Markowitz_bnds == 0;
 bnds = 0;
else;
 bnds = _Markowitz_bnds;
endif;
{alpha,u1,u2,u3,u4,retcode} = QPROG(sv,Q,R,A,B,C,D,bnds);
if retcode /= 0;
retp(error(0),error(0),error(0));
endif;
portreturn = alpha'mu;
portrisk = sqrt(alpha'Mcov*alpha);
retp(alpha,portreturn,portrisk);
endp;
```

At the command prompt type:
`lib marko marko.dec`
enter
`lib marko marko.src`
enter

This will create the library file MARKO.LCG that appears like

```
c:\ wingauss \ src \ marko.dec
 _markowitz_a :  matrix
 _markowitz_b :  matrix
 _markowitz_c :  matrix
 _markowitz_d :  matrix
 _markowitz_bnds :  matrix

c:\ wingauss \ src\ marko.src
 markowitz :  proc
```

These files must be located along SRC_PATH. The library file must be on LIB_PATH. With these files in place, the autoloader will be able to find everything needed to run the following program

```
new;
library coripe,pgraph;
#include portdata.txt;
```

The file portdata.txt should contain the following instructions

```
let mu[6,1] = 15.852
 14.262
 31.336
 25.775
 50.228
 14.842;
let sigma[6,1] = 37.215
 41.773
 33.165
 62.009
 60.720
 23.757;
```

```
   let Mcor[6,6] = 1.000 0.944 0.146 0.231 0.379 0.258
    0.944 1.000 0.109 0.239 0.413 0.223
    0.146 0.109 1.000 -0.169 -0.229 0.691
    0.231 0.239 -0.169 1.000 0.882 -0.256
    0.379 0.413 -0.229 0.882 1.000 -0.284
    0.258 0.223 0.691 -0.256 -0.284 1.000;
   Mcov = Mcor.*sigma.*sigma';

   /* the file must located along the SRC_PATH (SRC subdirectory)
*/

   weights = {};
   retur = {};
   risk = {};
   phi = 1;
   do until phi > 6;
    {alpha,rp,sigmap} = markowitz(mu,Mcov,phi);
    retur = retur~rp;
    risk = risk~sigmap;
    weights = weights~alpha;
    phi = phi + 1;
   endo;
   cls;
   output file = markow1.out reset;
   format /rd 10,3;
   print ''==================================================''
;
   print '' RISK AVERSION '' ;
   print '' ------------------------------------------------
'' ;
   print seqa(1,1,6)';
   print ''==================================================''
;
   print '' PORTFOLIO MEAN RETURN '' ;
   print '' ------------------------------------------------
'' ;
   print retur;
```

```
   print ''=====================================================''
;
   print '' PORTFOLIO RISK '' ;
   print '' -----------------------------------------------------
'' ;
   print risk;
   print ''=====================================================''
;
   print '' PORTFOLIO COMPOSITION '' ;
   print '' -----------------------------------------------------
'' ;
   print weights;
   output off;
   retur = {};
   risk = {};
   phi = 1;
   do until phi > 500;
    {alpha,rp,sigmap} = markowitz(mu,Mcov,phi);
    retur = retur | rp;
    risk = risk | sigmap;
    phi = phi + 1;
   endo;
   _pdate = '''';
   _pnum = 2;
   _paxht = 0.2;
   _pnumht = .2;
   _ptitlht = .2;
   title(''Efficient Frontier'');
   xlabel(''Portfolio Risk'');
   ylabel(''Portfolio Mean Return'');
   xy(risk,retur);
```

```
/*
** > QProg
**
** Purpose:  solves the quadratic programming problem
**
** Format:
** { x,u1,u2,u3,u4,ret } = QProg( start,q,r,a,b,c,d,bnds );
**
**
** Input:  start Kx1 vector, starting values
**
** q KxK matrix, model coefficient matrix
**
** r Kx1 vector, model constant vector
**
** a MxK matrix, equality constraint coefficient matrix
** if no equality constraints in model, set to zero
**
** b Mx1 vector, equality constraint constant vector
** if set to zero and M > 1, b is set to Mx1 vector
** of zeros
**
** c NxK matrix, inequality constraint coefficient matrix
** if no inequality constraints in model, set to zero
**
** d Nx1 vector, inequality constraint constant vector
** if set to zero and N > 1, d is set to Mx1 vector
** of zeros
**
** bnds Kx2 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
** are set to the plus and minus _qpbgnum
**
** Output:  x Kx1 vector, coefficients at the minimum of the function
**
** u1 Mx1 vector, Lagrangian coefficients of equality constraints
**
```

```
** u2 Nx1 vector, Lagrangian coefficients of inequality constraints
**
** u3 Kx1 vector, Lagrangian coefficients of lower bounds
**
** u4 Kx1 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
**
** Globals:  _qprog_maxit - scalar, maximum number of iterations,
** default = 1000
**
** Remarks:  QProg solves the standard quadratic programming problem:
**
** minimize 0.5 * x'Qx - x'R
**
** subject to constraints,
**
** Ax = B
** Cx >= D
**
** and bounds,
**
** bnds[.,1] <= x <= bnds[.,2]
**
*/
```

9.3 OPTMUM Library

Example

$$f(x,y) = (x-3)^2 + (y-x-3)^2 + xy$$

vector representation

$$f(X) = (X_1-3)^2 + (X_2-X_1-3)^2 + X_1X_2$$

with

$$X = \begin{bmatrix} x & y \end{bmatrix}'$$

```
new;
library optmum;
proc function(x);
local x1,x2,y;
x1 = x[1];
x2 = x[2];
y=(x1-3)^2+(x2-x1-3)^2+x1*x2;
retp(y);
endp;


sv = 1 | 1;
output file = resopt.out reset;
{xmin,fmin,gmin,retcode} = optmum(&function,sv);
call optprt(xmin,fmin,gmin,retcode);
output off;
```