Genstat®

Syntax and data management

www.vsni.co.uk

VSNi

The Guide to the Genstat® Command Language (Release 22)

Part 1: Syntax and Data Management

Genstat is developed by VSN International Ltd, in collaboration with practising statisticians at Rothamsted and other organisations in Britain, Australia, New Zealand and The Netherlands.

Published by: VSN International, 2 Amberside, Wood Lane,
Hemel Hempstead, Hertfordshire HP2 4TP, UK
E-mail: info@genstat.co.uk
Website: http://www.genstat.co.uk/

First published 2000, as *The Guide to Genstat*
This edition published 2022, for Genstat Release 22

Citation: VSN International (2022). *The Guide to the Genstat Command Language (Release 22), Part 1 Syntax*. VSN International, Hemel Hempstead, UK.

Preface

Modern biological research began at Rothamsted in 1843, when Sir John Bennet Lawes started the Broadbalk wheat experiment which is now the world's longest-running field experiment. Rothamsted also pioneered the application of statistics in biological research when Sir Ronald Fisher was appointed in 1919 to study the accumulated results of Broadbalk and its many subsequent experiments. Fisher soon realised the need for improved statistical techniques over the whole range of agricultural and biological research, and the groundwork for modern applied statistics was laid by him and his colleagues during the 1920s and 1930s.

Statistical computing began at Rothamsted when Fisher's successor Frank Yates obtained an Elliot 401 computer – one of the first computers to be used away from its manufacturing base, and one of the first to be used for statistical work. This extended the tradition, started by Fisher, of conducting statistical research to solve real problems arising from biological research. The resulting new methods could now be implemented in the Rothamsted statistical programs to enable them to be used more effectively in practice. The development of Genstat at Rothamsted began in 1968, when John Nelder took over from Yates as Head of Statistics. Roger Payne took over leadership of Genstat when Nelder retired in 1985.

Genstat began to be distributed outside Rothamsted during the 1970's and, in 1979, its distribution was taken over by the Numerical Algorithms Group (NAG), one of the world's oldest technical computing companies. More recently Genstat has been developed and marketed by VSN International (VSNi). VSNi was formed in 2000 as a spin-off company from Rothamsted and NAG. This brought together the Genstat development group from Rothamsted with the statistical commercialization group from NAG to provide a stronger collaboration of research and development with sales and marketing. However, the development group retain their close links with the research community through the continuing links between VSNi and Rothamsted. So users benefit from the rigorous quality control required in a commercial setting, while still retaining the underlying excitement from the research environment.

Acknowledgements

The development of Genstat has involved many people over the years. In particular, special thanks are due to:

A.E. Ainsley, N.G. Alvey, D.B. Baird, C.F. Banfield, R.I. Baxter, K.E. Bicknell, I.C. Channing, B.R. Cullis, P.G.N. Digby, A.N. Donev, M.F. Franklin, A.R. Gilmour, A.I. Glaser, J.C. Gower, S.A. Harding, T.J. Hastie, S.K. Haywood, A.F. Kane, A. Kobilinsky, W.J. Krzanowski, P.W. Lane, S.D. Langton, P.J. Laycock, P.K. Leech, J.H. Maindonald, G.W. Morgan, D.A. Murray, J.A. Nelder, A. Papritz, H.D. Patterson, R.W. Payne, D.L. Robinson, G.J.S. Ross, P.J. Rowley, H.R. Simpson, D.M. Soutar, R.J. Tibshirani, A.D. Todd, R. Thompson, G. Tunnicliffe Wilson, L.G. Underhill, P.J. Verrier, R.W.M. Wedderburn, S.J. Welham, R.P. White, R. Webster and G.N. Wilkinson.

`

Contents

1 Introduction, syntax and terminology

Genstat is a very general computer program for statistical analysis. All the usual analyses are readily available using the commands in the language, as are many advanced techniques. Furthermore, if you are running Genstat on a PC with Microsoft® Windows™, you will find that almost anything you want will be available by using a menu. The menus in Genstat *for Windows* operate by automatically generating and running "scripts" of Genstat commands. The commands can be saved to provide an "audit trail" of your analysis. They record exactly what you have done, and can be run again later to reproduce the analysis. So you may be interested to know more about the command language in order to understand the scripts. You may then find that you can save time by modifying an existing script to generate a new analysis. For example, you may want to run a script several times with different datasets by defining a *loop* (see Section 5.2.1).

As you learn more about the language, you will see that Genstat is not just a collection of pre-programmed commands for selecting from fixed recipes of available analyses. You can use the command language to write your own programs to cover the occasions when the standard analyses do not give exactly what you want, or when you want to develop a new technique. The commands give you complete control over what is printed, and virtually anything that can be printed from an analysis can be stored in a Genstat data structure (Chapter 2) and used as input for another command. So any Genstat analysis can be used in the construction of a new technique. Most users will need to do this only occasionally, since the standard facilities in Genstat are extremely comprehensive. However, the ability to extend Genstat removes the temptation, that occurs with some other packages, to use an inappropriate or approximate technique when an unusual set of data has to be analysed. Programs can be formed into procedures, to simplify their future use or to make them easily available to other users (see Chapter 5).

This book, Part 1 of the *Guide to the Genstat Command Language*, describes the syntax of the Genstat language, and the facilities that Genstat provides for input and output, data manipulation, calculations and programming. Genstat's extensive statistical facilities are described in Part 2. References below to Part 2 are prefixed by "2:". So, for example, 2:3.1 refers to Part 2 Section 3.1, while 3.1 refers to Section 3.1 in this book.

This chapter describes the basic rules, terminology and conventions of the Genstat language. These are common to all Genstat commands. So, once you understand these, you can access any of the facilities that Genstat provides. Section 1.1 illustrates the concepts with simple examples, while the later sections contain the formal definitions.

The example output in the *Guide* is produced in Genstat's *plain-text* style to distinguish it more clearly from the textual descriptions. However, as an alternative, you can generate output in *formatted* styles, including RTF, HTML and LaTeX; see Sections 3.3.1 and 3.3.4. These formatted styles also allow you to use subscripts, superscripts and Greek and mathematical symbols; see Section 1.4.2.

1.1 Running Genstat

The Genstat language is designed for *interactive* use. In other words, you can give commands to be executed by Genstat one at a time, so that you can see the result of each command before you give the next one. Genstat can also be used in *batch mode*. Here you construct a file of commands and supply them to Genstat all at once. Batch mode may be more convenient for routine work, or when carrying out complicated analyses that may take a lot of computer time.

In Genstat *for Windows*, you can run Genstat interactively by typing commands into a text window, and executing them one line at a time using the Run menu on the menubar. Batch mode can be achieved either by submitting all the commands in a text window at once (again using the Run menu), or by running the Submit File menu (also accessible from the Run menu).

1.1.1 Interactive mode

First, we show how to run Genstat interactively. On many types of computer, you can start Genstat running interactively by giving the instruction

```
      Genstat
```

If this does not work, you will need to refer to the local documentation to find out how to get started. For example, on workstations or mainframes, information may be available using the Help system on the computer. In any case, though, a copy of the relevant instructions is supplied to everyone who buys Genstat. For example, full details of how to run Genstat *for Windows* are given in the book *Genstat for Windows Introduction* and the on-line tutorials that accompany it. Genstat *for Windows* offers considerable flexibility over the ways in which commands are supplied, which are all described in the *Introduction*. In this Guide, however, we cover only the simpler situations that apply to all types of computer.

Genstat starts with some initial information telling you what version of Genstat and its Procedure Library (see Section 5.3.1) you are using, and in most implementations (but not PC Windows) it then displays a new line prefixed by the command prompt

```
      >
```

You can now type in your first command. Generally the command will use a Genstat *directive* (our term for a standard command). You can also use *procedures*. These are self-contained sets of commands, like a sub-program in the Genstat language. Genstat has a library of standard procedures, and you can also write your own. Details are given in Section 5.4, but all you need to know for now is that the rules for using procedures are exactly the same as those for using directives. Full details are given later in this chapter, but we explain the basic ideas now.

We introduce the rules in the context of a directive called `PRINT`. This displays data: either on the screen when you are running interactively, or in an output file if you are running in batch (Section 1.1.2). For example, you can give the simple command

```
      PRINT 1
```

to display a single set of data: the number 1. The display looks like this:

```
> PRINT 1

      1.000

>
```

This is clearly not a very useful operation, because you already know what the set of data is, and because it consists only of a single number; however, this will be generalized in Section 1.2. In the meantime, you can see that the directive name, `PRINT`, is like a command verb which instructs Genstat to do something, and the number `1` is like the object of the command. The object is called the *primary parameter* of the command.

The `PRINT` directive works with sets of data. You can make it work with several sets of data at once by giving a *list*; for example, the command

```
      PRINT 1,2
```

has two sets, each containing one number. The display is:

```
> PRINT 1,2

      1.000       2.000

>
```

In Genstat, lists are always constructed using commas. You must not use just spaces; for example, the command

```
PRINT 1 2
```

will be faulted, producing an error message:

```
> PRINT 1 2

******** Fault (Code SX 12). Statement 1 on Line 3
At...   PRINT 1 \2\:
Incompatible adjacent elements (e.g. comma missing)

>
```

You can use spaces as well as commas if you want. So the following command is acceptable:

```
PRINT 1 ,    2
```

You will have noticed that `PRINT` commands lay out the data in a tabular form, choosing an appropriate number of decimal places for numbers. By default, a single number is displayed with four significant digits. Also, sets of data with compatible shape are laid out in *parallel*: that is, side-by-side. If you do not want this default display, there is a range of *options* for modifying it. For example, the command

```
PRINT [SERIAL=yes] 1,2
```

displays the two numbers in *serial* form rather than in parallel: that is, the number 1 by itself, and then the number 2:

```
> PRINT [SERIAL=yes] 1,2

       1.000

       2.000

>
```

Most Genstat directives and procedures have options like this to control the way in which they work. The options must always be given in square brackets following the directive or procedure name and preceding the parameters, if any. Options have the form *name*=*setting*, where here the name is `SERIAL` and the setting is `yes`. If you set several options, you must separate them with a semi-colon, as in

```
PRINT [SERIAL=yes; INDENTATION=10] 1,2
```

This command would indent the output by 10 characters so that, if you arrange to send the display to a printer, you could rely on having a clear margin on the paper, perhaps for binding.

Most Genstat directives and procedures also have *auxiliary parameters* which control the way the command works. For example, the command

```
PRINT 1,2; DECIMALS=0,1
```

gives the following display:

```
> PRINT 1,2; DECIMALS=0,1

           1         2.0

>
```

The `DECIMALS` parameter specifies how many decimal places to display for each set of data. The essential difference between an option and an auxiliary parameter is that an option specifies a modification once and for all for the command: an auxiliary parameter specifies a modification that may be different for each of the sets of data in turn. The setting of the `DECIMALS` parameter above, `0,1`, is matched item by item with the setting of the primary parameter, `1,2`. This distinction applies to all Genstat commands.

The setting of an auxiliary parameter is otherwise like that of an option, with the form *name=setting*, and with the semi-colon separator between successive parameters. The primary parameter itself has a name, except when there are no auxiliary parameters. So you could actually give the command:

```
    PRINT STRUCTURE=1,2; DECIMALS=0,1
```

However, if you specify the primary parameter first in a command, its name can always be omitted.

You can abbreviate directive and procedure names to the first four characters. Names of options and parameters can also be abbreviated to four characters, and sometimes further. The full abbreviation rules are described in Section 1.7.

You can end your interactive run of Genstat using the `STOP` directive:

```
    STOP
```

Genstat keeps a log of an interactive session on the computer. In Genstat *for Windows*, this is in a window called the Input Log. On other Genstat implementations, the log will be stored in a computer file. The name of the file will depend on the type of computer, but is given in the local documentation. The file contains all the commands and data values that you have typed, up to and including `STOP`. This allows you to check what you have done, or to keep a record for future reference, though on some computers it may be necessary to copy the file to avoid it being overwritten by the log of the next session. It also allows you to modify the commands using an editor and then to execute them all again using batch mode.

1.1.2 Batch mode

We now illustrate the use of Genstat in batch. Suppose that we have taken nine samples of polluted soil and measured the amount of zinc in each of them, and that we wish to calculate some simple summary statistics of the amounts of zinc in the samples. First of all we need to set up a file on the computer containing all the commands for Genstat to process. You can do this using any of the facilities for creating files on your computer, such as a text editor (or, for example, by opening a new text window in Genstat *for Windows*). The input file, below, shows the commands required for the zinc example.

```
    VARIATE [NVALUES=9] IDENTIFIER=Zinc
    READ STRUCTURE=Zinc
    164.2 160.6 163 166 159.8 163.9 161 161.3 165.8 :
    SCALAR IDENTIFIER=Zmed,Zvar
    CALCULATE Zmed = MEDIAN(Zinc)
    CALCULATE Zvar = VARIANCE(Zinc)
    PRINT STRUCTURE=Zmed,Zvar; DECIMALS=1,2
    STOP
```

You then give an instruction to the computer to run Genstat with that file attached as input and another file allocated to receive the output. The command should also allow you to control aspects like the style of output. In Genstat *for Windows*, this can be done using the Submit File menu. Details of the necessary instruction for other implementations will again be in your local documentation, but it will probably be something like this:

```
    Genstat input-filename,output-filename
```

The output that would be generated for the zinc example is shown below.

Example 1.1.2

```
   1  VARIATE [NVALUES=9] IDENTIFIER=Zinc
   2  READ STRUCTURE=Zinc

  Identifier   Minimum      Mean   Maximum    Values   Missing
        Zinc     159.8     162.8     166.0         9         0

   4  SCALAR IDENTIFIER=Zmed,Zvar
   5  CALCULATE Zmed = MEDIAN(Zinc)
   6  CALCULATE Zvar = VARIANCE(Zinc)
   7  PRINT STRUCTURE=Zmed,Zvar; DECIMALS=1,2

       Zmed        Zvar
      163.0        5.22
```

Notice that in batch mode Genstat echoes the input lines, each one prefixed by the number of that line in the input file, whereas it does not do so when running interactively. (This is Genstat's default action, but it can be altered by using the `SET` directive, as described in Section 5.6.1, or by using the Options menu of Genstat *for Windows*.)

A sequence of commands to Genstat, like those used in this example, is called a Genstat *program*. Each command is known as a Genstat *statement*, and requests Genstat to perform some sort of action. The statement may use either a *directive* (our term for a standard command) or a *procedure* (a self-contained set of statements, like a sub-program in the Genstat language; see Section 5.4). However, the syntax is the same in either case.

The program in Example 1.1.2 first declares a *data structure* with the *identifier* (or name) `Zinc` to store the amounts of zinc in the samples. Several different types of data structure are available in Genstat. This one is known as a *variate*, and can be defined using the `VARIATE` directive. Variates are used to store a list of numbers, in this case of length nine. Example 1.1.2 also has two *scalar* data structures, `Zmed` and `Zvar`, each of which stores a single number.

The example shows that values can be assigned to data structures in various ways. The values for `Zinc` are input in line 2 using the `READ` directive (see Section 3.1.2). The values for the scalars are assigned from the results of two calculations, using the `CALCULATE` directive (lines 5 and 6). As you will see in Section 4.1.5, `CALCULATE` is also able to define data structures automatically, and `Zmed` and `Zvar` are declared automatically as scalars so that they can store the single value generated by each calculation. The `PRINT` directive in line 7 displays the values of `Zvar` and `Zmed`, similarly to the way in which the numbers 1 and 2 were printed in Section 1.1.1. Finally (but not shown in the example), you can use the `STOP` directive to indicate the end of the program, whether you are running interactively or in batch.

1.2 Genstat programs

As you have seen, a *program* consists of a series of instructions to Genstat, or *statements* in our terminology. Before describing the general syntax of statements (1.3 to 1.8), we shall summarize some of the basic things that they can do.

1.2.1 On-line help

Help is available on-line while you are running Genstat. Many implementations (especially Genstat *for Windows*) allow you to access a browseable help file, with hot links to enable you to locate topics of interest. Genstat *for Windows* provides context-sensitive help as well. To access this, you put the cursor into the word of interest, or the first word of the phrase of interest, and then press the F1 key.

In Genstat *for Windows*, you can open the help file by clicking in the Contents and Index line in the Help menu on the menubar. An alternative (which also works in other Genstat implementations) is to type the command

```
HELP
```

1.2.2 Declarations

A statement specifying the type and identifier of a data structure is called a *declaration*. Declarations can be explicit or implicit. An example of an explicit declaration is the VARIATE statement in Example 1.1.2. Examples of implicit declarations are shown in the CALCULATE statements: the particular calculations done here produce a single-valued result, and so implicitly define scalar structures.

Other kinds of calculation produce other kinds of results, thus implicitly defining other kinds of structures. Implicit declarations are called *default declarations*: the rules for these are described in this Guide at the same point as the directives that make them.

1.2.3 Assigning values

You can define data values in the declaration itself: for example, the Zinc values could be defined by

```
VARIATE [VALUES= 164.2, 160.6, 163, 166, 159.8,\
         163.9, 161, 161.3, 165.8] Zinc
```

(The symbol \ continues the statement onto a second line; see Section 1.4.6.) Alternatively you can read the values, as was done for Zinc in Example 1.1.2, or you can derive values as the results of calculations, as for the scalars Zmed and Zvar.

Later you will see that statistical analyses too can derive values to be assigned to data structures. You will see also that the data can be read from files other than the main input file (that is, instead of listing the data immediately after the READ statement).

1.2.4 Calculations

Calculations can be done with many kinds of data structure. Genstat contains many flexible tools for analysing data, ranging from the simplest (such as taking means) to the advanced (like nonlinear regression). But the essential point is that they do calculations with data held in data structures and referred to by their identifiers.

1.2.5 Printing

Many of the statistical directives in Genstat produce their own output. For example, the ANOVA directive will produce an analysis-of-variance table, tables of means, standard errors and so on. But you may often want to produce output of your own. PRINT is merely one way of doing that. Genstat can also for example produce tables, point or line plots and histograms.

1.2.6 Statements

The one thing that all these features of Genstat have in common is that you get access to them by means of the statements that make up a program. (The menus in Genstat *for Windows* operate by automatically forming statements of instructions for Genstat; these can be seen in the Input Log.)

All statements have the same rules of syntax: first you give the name of the directive or procedure that you wish to use, then perhaps some options, and then usually some parameters. The names are intended to be natural, or to refer naturally to common statistical techniques. Appendix 1 lists the directives in Release 13.1, together with the procedures in Release PL21 of the standard library which is the one included with Release 13.1.

Options are enclosed in square brackets, as in the declaration of the variate Zinc in Example 1.1.2. Each option has a name (for example NVALUES), and you can give it an appropriate *setting* (for example 9 here): the general form for setting an option is "name=setting". Another example is the INDENTATION option in the following statement:

```
PRINT [INDENTATION=7] STRUCTURE=Zmed,Zvar
```

This would indent the printed values by seven spaces from the left-hand margin of the page. Some options have default settings, that are the settings assumed by Genstat if you do not specify any explicitly. For example, the default for indentation is zero (the values are printed from the left margin). Conversely, the VARIATE directive has an option called VALUES, which can be used to define the values in a variate when it is declared; this has no default, and if it is omitted no values are defined.

Parameters are set in a similar way to the options, coming after the close of the square brackets (if any). One parameter that is nearly always part of a statement is a list of identifiers or an expression on which the statement is to operate. Some directives allow no more than this – for example CALCULATE; in such cases, there is no name for the parameter. Other directives have several parameters. For example, in the PRINT statement in Example 1.1.2 two parameters are set: STRUCTURE and DECIMALS. Sometimes the names of the parameters can be left out. The full rules that Genstat then uses to determine which parameter is being set are described in Section 1.7.1; but the simplest rule is that if no name is included for the first parameter given in a statement, Genstat takes this as the setting for the main parameter of the statement. As you will have seen in the example in Section 1.1.1, the main parameter of the PRINT lists the structures whose values are to be printed.

If more than one parameter is set, each one must be separated from the next by a semicolon. The same rule applies if several options are to be set. The lists specified for each of the parameters are taken by Genstat in parallel, so the statement

```
PRINT [INDENTATION=7] Zmed,Zvar; DECIMALS=1,2
```

will print Zmed with one decimal place and Zvar with two, and all this information will be indented by seven spaces. The parameters after the main parameter are thus used to supply further information for each item in the main list. Conversely, options supply information that applies to all the parameters in the list.

In this Guide we generally give names of directives, options and parameters in capitals and in full. But small letters and abbreviations can be used if you prefer. In particular, it is always enough to give four characters, but option and parameter names can often be abbreviated beyond that (1.7.1).

1.2.7 Punctuation

Items in lists are separated by commas: see, for example, the list of identifiers in the PRINT statement in Example 1.1.2. Option settings are separated from each other by semicolons, as are parameter settings.

The usual way of ending a statement is with the carriage-return key, usually labelled <RETURN> or ↵ on the keyboard. But you can also end with a colon, and thus get several statements on one line. The continuation symbol \ allows you to continue the statement onto the next line.

1.2.8 Comments

You can put comments into your programs to help other people to understand them, or to help you remember them if you need to reuse them later on. The series of comments can then give a running description of what the program is doing. You tell Genstat that you are making a comment by using the double-quote character ("); notice that this is not the same as two single quotes (''). In Example 1.1.2, we could add the comment

```
"This program calculates some simple statistics
to summarize the amount of zinc in the samples."
```

You can type anything you like between the double quotes; Genstat simply ignores it. In longer programs you might want to put comments at several places in the program, to describe what different sets of statements are doing. In an interactive run Genstat will add the double-quote character to the prompt to remind you when you are in the middle of a comment, i.e.

```
    ">
```

while in a batch run the line number is prefixed by a minus sign.

1.3 Characters

Sections 1.3 to 1.8 contain a rigorous definition of the Genstat language, starting with the simple aspects and moving gradually to the more complicated. You may prefer not to read these sections immediately, but to return to them when you need to know more details of the rules; however, it will probably be useful to read about the conventions used in this Guide, given in Section 1.9. There is much cross-referencing among Sections 1.3 to 1.8, and there are also references forward to the rest of the Guide.

The characters in Genstat statements are classified as in Sections 1.3.1 to 1.3.7.

1.3.1 Letters

A *letter* is any of the alphabetic characters `A`, `B`, up to `Z`, `a`, `b`, up to `z`, the underline character (_) and the percent character (`%`).

1.3.2 Digits

A *digit* is one of the numerical characters 0, 1, 2, up to 9.

1.3.3 Simple operators

These occur in arithmetic *expressions* or in the *formulae* that define statistical models. The *simple operators* are:

```
    +  -  *  /  .  =  <  >
```

Equals (=), less than (<) and greater than (>) occur only in expressions (1.6.2). Dot (.) occurs only in formulae (1.6.3).

The meanings of the simple operators, and of the *compound operators* made up of more than one character, are given in 1.4.6.

1.3.4 Brackets

There are two kinds of *bracket*.

Round brackets `(` or `)` are used in lists (1.5) and expressions (1.6.2), and to enclose the arguments of functions (1.6.1).

Square brackets `[` or `]` enclose option settings, and are also used for suffixed identifiers (1.5.3). Left curly bracket `{` is synonymous with left square bracket `[`, and right curly bracket `}` with right square bracket `]`; these provide alternatives if square brackets are unavailable on your keyboard.

1.3.5 Punctuation symbols

Punctuation is used to separate different components of statements.

The *space* character can be used to improve the layout and readability of your programs. Statements use *free format*: that is, there may be any number of spaces between items; items are defined in 1.4. Spaces can be left out altogether if the items are already separated by another punctuation symbol, by a bracket, by an operator, or by a special symbol (1.3.6). Most keyboards have a tab key (`<TAB>`), which has the effect of inserting spaces before subsequent characters on the terminal screen. Genstat treats the tab character as a synonym of space everywhere except within strings (1.5.2) and comments (1.2.8), or if reading in fixed format when it is treated as a fault (3.1.7).

Comma (`,`) is used to separate items in lists; lists are described in 1.5.

Equals (=) separates an option name or a parameter name from the list of settings. This character can thus have two meanings (separator or operator) but it will always be clear which is intended from the context.

Semicolon (`;`) is used to separate one list from another.

Colon (`:`) marks the end of a statement.

Newline is obtained by pressing the carriage-return key (`<RETURN>`). It is another way of marking the end of a statement. (Note, however, that the `SET` directive can be used to request that newlines be ignored; see 5.6.1).

Single quote (`'`) marks the start and finish of a string (1.5.2). On many computer terminals, there are two kinds of quote (` and '); these are synonymous.

Double quote (`"`) marks the beginning and end of a comment (1.2.8).

1.3.6 Special symbols

Some characters have more specialized meanings; details of the ways in which they are used are given later in this chapter.

Ampersand (`&`) indicates that the directive name or procedure name from the previous statement is to be repeated, together with any option settings that are not explicitly changed (1.7.4).

Asterisk (`*`) is used to denote a missing value (1.4.5); this is another character with two meanings, missing value or operator, which again are easily distinguished by context.

Backslash (`\`) indicates that a statement is continued on the next line (1.7).

Dollar (`$`) is used to define subsets of a data structure. The dollar is followed by a list enclosed in square brackets, which specifies the contents of the subsets (1.5.3).

Exclamation mark (`!`) introduces an *unnamed* data structure (1.4.3). The vertical bar (|), available on some keyboards, is synonymous with exclamation mark.

Hash (`#`) is followed by the identifier of a data structure whose values are to be inserted at the current point of the program (1.5.4). It can also indicate the default setting of an option (1.7.3). On some keyboards and printers, `#` is replaced by `£`.

Tilde (`~`) is used to introduce a typesetting command within a string (1.4.3).

1.3.7 Non-ASCII characters

Your computer may also allow you to define additional (non-ASCII) characters, such as accented letters and Chinese, Korean or Thai characters. These can be used in Genstat strings (1.4.2) and text structures (2.3.2).

1.4 Items

A Genstat statement can contain various pieces of information; we shall call these *items*. There are six kinds of item, illustrated in these statements that calculate and print the area of a circle:

```
    CALCULATE Area = 3.142 * Radius**2
    PRINT [IPRINT=*] 'The area is',Area
```

The option setting `IPRINT=*` stops the name of the data structure being printed.

The words `CALCULATE`, `PRINT` and `IPRINT` are *system words*, whereas `Radius` and `Area` are *identifiers* and the quoted characters make up a *string*. There are two *numbers* (`3.142` and `2`) and three *operators* (=, `*` and `**`). The asterisk inside the square brackets is a *missing value*. The statements contain some other characters, which separate the items: these are the square brackets, the equals sign in the options and the comma between `'The area is'` and `Area`.

1.4.1 Numbers

A *number* represents quantitative information, and in its simplest form consists of digits only. For example,

```
0    245609
```

A number can also have a *sign* (+ or -) and a decimal point (.).

```
-2     4.5    +33.     -.2
```

However, a number must not contain any commas. Thus you must write one thousand as `1000` not `1,000`.

To avoid lots of zeroes in large or small numbers, you can use an *exponent*. For example `2E-20` means 2×10^{-20}. Another example is `2D-5` which means 0.00002. `D` and `E` have the same meaning, and can also be replaced by `d` or `e`: these four are all called *exponent codes*. In general, a number can have the form

```
xEy
```

which means that the number *x* is to be multiplied by `10` to the power *y*. The number *x* can have a sign, as can the exponent *y*. There must not be any spaces between *x* and the exponent code, nor between a sign and the exponent. But there can be spaces between the exponent code and the exponent: for example `2d  -5` again means 0.00002.

Numbers can also be used within Genstat to represent dates and times. The time is contained in the decimal part of the number, and represents the time during the day. So, for example, 12 noon is stored as 0.5, and 6 p.m. as 0.75. The date is contained in the integer part of the number, and represents the number of days of the date since the *basedate*. Genstat bases this on the Gregorian calendar, so day 1 was 1st March 1600, 31st December 1999 is stored as 146037, 1st April 2000 is stored as 146129, and so on. The difference between two dates and times is thus a time duration, so dates and times can be analysed and manipulated like any other data. The directives that define numerical data structures all have a parameter `DREPRESENTATION` to define a format to be used to display the number as a date and time whenever the data structure concerned is printed (2.1.5). Alternatively, the format can be specified at the time that the structure is printed, using the `DREPRESENTATION` parameter of the `PRINT` directive itself (3.2.1).

1.4.2 Strings

A *string* is a piece of textual information. Some examples are

```
apple;
five apples;
5 apples.
```

The spaces and punctuation here are part of the string. Important uses of strings are to form the values of text data structures (2.3.2) and to annotate output from a program (3.2.1).

More formally, we define a string to be a series of characters conveying textual information. In most places *quoted strings* are required: there, the characters are placed between single quotes (`'`); for example

```
'apple'
```

Quoted strings may contain any of the characters available on your computer.

In some places an *unquoted string* can be used. This must have its first character as a letter and all its characters as letters or digits.

Upper-case and lower-case letters are distinct within strings; so the strings `Apple` and `apple` are not the same.

If you want to put a single quote itself into a quoted string, you must put it in twice; otherwise Genstat thinks the string is ending. For example

```
'don''t do that'
```

will be interpreted as

```
    don't do that
```

Similarly, a quoted string cannot contain a double-quote character on its own, because this is interpreted inside a string as the start of a comment (1.2.8): a comment inside a string is not interpreted as part of the string but is ignored. So to include a double quote in the string, you need to put two double quotes.

A continuation symbol (\) on its own in a quoted string continues the string onto the next line. However, a pair of backslash characters is interpreted as a single appearance of that character. For example

```
    'C:\\Examples\\Regress.gen'
```

is interpreted as

```
    C:\Examples\Regress.gen
```

If a quoted string contains a newline (`<RETURN>`) that does not follow an unduplicated continuation symbol, then it becomes a *string list* (1.5.2), unless you have used the `SET` directive to specify that newlines are to be ignored (5.6.1).

Strings can contain typesetting commands to represent Greek and mathematical symbols. The commands are converted automatically by Genstat to match the style of output (HTML, LaTeX, plain-text or RTF). They are all introduced by the character tilde (~). So, to use tilde as an ordinary character, you need to specify the special symbol `~{~}` as defined below.

If Genstat finds a mistake in the syntax of a command, it will not issue a failure diagnostic but will output the remainder of the string (including any commands) as plain text. (So programs containing tilde characters, written before these commands were introduced in Release 9, should continue to work as before.) The following commands define Greek characters and various special and mathematical symbols.

~{~}	tilde symbol; also see `~{tilde}`
`~{alpha}`	Greek character alpha
`~{beta}`	Greek character beta
`~{gamma}`	Greek character gamma
`~{delta}`	Greek character delta
`~{epsilon}`	Greek character epsilon
`~{varepsilon}`	Greek character epsilon (variant)
`~{zeta}`	Greek character zeta
~{eta}	Greek character eta
`~{theta}`	Greek character theta
`~{vartheta}`	Greek character theta (variant)
`~{iota}`	Greek character iota
`~{kappa}`	Greek character kappa
`~{lambda}`	Greek character lambda
`~{mu}`	Greek character mu
`~{nu}`	Greek character nu
`~{xi}`	Greek character xi
`~{omicron}`	Greek character omicron
`~{pi}`	Greek character pi
`~{varpi}`	Greek character pi (variant)
`~{rho}`	Greek character rho
`~{varrho}`	Greek character rho (variant)
`~{sigma}`	Greek character sigma
`~{varsigma}`	Greek character sigma (terminal version)
`~{tau}`	Greek character tau
`~{upsilon}`	Greek character upsilon
`~{phi}`	Greek character phi

`~{varphi}`	Greek character phi (variant)
`~{chi}`	Greek character chi
`~{psi}`	Greek character psi
`~{omega}`	Greek character omega
`~{bull}` or `~{bullet}`	bullet
`~{cdot}`	decimal point; also see `~{middot}`
`~{div}` or ~{divide}	divide symbol
`~{gg}`	">>" symbol; also see `~{raquo}`
`~{laquo}`	"<<" symbol; also see `~{ll}`
`~{ll}`	"<<" symbol; also see `~{laquo}`
`~{middot}`	alternative way of specifying a decimal point; also see `~{cdot}`
`~{minus}`	minus symbol
`~{plusminus}`	"+ or minus" symbol; also see `~{pm}`
`~{pm}`	"+ or minus" symbol; also see `~{plusminus}`
`~{raquo}`	">>" symbol; also see `~{gg}`
`~{sqrt}`	square-root symbol
`~{oplus}`	+ within circle
`~{ominus}`	minus symbol within circle
`~{otimes}`	multiply symbol within circle
`~{oslash}`	slash symbol within circle
`~{odot}`	dot within circle
`~{tilde}`	tilde symbol; also see ~{~}
`~{times}`	multiply symbol
`~{break}`	starts a new line

The character definitions (within the curly brackets) can be abbreviated. Genstat checks through the possibilities, in the order defined above, until it finds the first match. Greek characters in capital letters can be obtained by beginning the name of the character with a capital letter, for example `~{Sigma}`; subsequent capital letters are irrelevant.

The style of font can be changed to bold or italic.

`~bold` or `~b`	introduces a sequence of bold characters; these must be placed within curly brackets and any spaces between `~bold` and the opening curly bracket are ignored. e.g. `~bold {Please note:}`
`~italic` or `~i`	introduces a sequence of italic characters; these must be placed within curly brackets and any spaces between `~italic` and the opening curly bracket are ignored. e.g. `~italic {Passer domesticus}`

You can also produce output in the same style as Genstat uses when it echoes commands in the output.

`~genstat` or `~g`	introduces some output in the style that Genstat uses to echo commands; it must be placed within curly brackets and any spaces between `~genstat` and the opening curly bracket are ignored.

You can define subscripts and superscripts (for example to define equations).

`~_`	introduces a subscript; if the subscript is a single character it can be placed immediately after _, otherwise it must be placed within curly brackets; any spaces between `~_` and the opening curly bracket are ignored.
`~^`	introduces a superscript; if the superscript is a single character it can be placed immediately after ^, otherwise

	it must be placed within curly brackets; any spaces between `~^` and the opening curly bracket are ignored.

You can use special characters in subscripts or superscripts, but fonts must be specified outside the subscript or superscript. For example:

`~i {x~_{i,j}}`	defines $x_{i,j}$,
`x~^ {2n}`	defines x^{2n}, and
`~i{x~_{i,j}}~^2`	defines $x_{i,j}{}^2$
`~b{X}~i{~_{i,j}}~^2`	defines $\mathbf{X}_{i,j}{}^2$.

For additional flexibility, you can specify output information in either HTML, LaTeX or RTF. This will be inserted only into output constructed by Genstat in the same style. You can also supply information to be included only in plain-text output (which may, for example, be your translation of the HTML, LaTeX or RTF information).

`~html` or `~h`	introduces a sequence of information in HTML; the information must be placed within curly brackets and any spaces between `~html` and the opening curly bracket are ignored.
`~latex` or `~l`	introduces a sequence of information in LaTeX; the information must be placed within curly brackets and any spaces between `~latex` and the opening curly bracket are ignored. The information may itself contain curly brackets. These are assumed to be paired according to the usual rules of LaTeX, except that any curly brackets preceded by the LaTeX escape character \ are ignored.
`~plain` or `~p`	introduces a sequence of information to be inserted only in plain-text output; the information must be placed within curly brackets and any spaces between `~plain` and the opening curly bracket are ignored.
`~rtf` or `~r`	introduces a sequence of information in RTF; the output must be placed within curly brackets and any spaces between `~rtf` and the opening curly bracket are ignored. The information may itself contain curly brackets. These are assumed to be paired according to the usual rules of RTF, except that curly brackets preceded by the RTF escape character \ are ignored.

1.4.3 Identifiers

An *identifier* is the name used to refer to a data structure. An *unsuffixed* identifier is made up of letters and digits, starting with a letter. For example,

```
    Cost    Yield2006    Yield2007
```

Any characters beyond the first 32 are ignored; so if you used the identifiers

```
  Production_recorded_during_month_1,
  Production_recorded_during_month_2
```

and so on, they would all refer to the same structure, namely

```
  Production_recorded_during_month.
```

By default, Genstat will treat capital letters as distinct from small letters. However, the `SET` directive (5.6.1) can been used to request that Genstat regards them as equivalent. `SET` can also request Genstat to use *short wordlengths* as always happened in releases before Release 4.2; only eight characters are then stored for identifiers, and the ninth and subsequent characters are ignored.

Identifiers can have suffixes; for example

```
    Yield[2006]
```

The suffix is enclosed in square brackets, and can be a number, a quoted string or another identifier. You can put spaces on either side of either of the square brackets.

A *suffixed* identifier is a value, or a set of values, of a pointer data structure (2.6). Thus `Yield[2006]` and `Yield[2007]` are two structures which are pointed to by the pointer structure `Yield`. (So a pointer is simply a structure that contains a list of other structures.) When you use a suffixed identifier Genstat will automatically define the necessary pointer (2.6). An identifier can also be *qualified* to specify a subset of its values (see Section 1.5.3).

The identifier is generally also used to label the data structure in output. However, Section 2.1.3 explains how you can use the `EXTRA` option of many declarations to associate an "extra" text of description with a data structure, and the `IPRINT` option to request that this description be used instead of the identifier.

If the data structure is supplying input to a command, and is going to be used only once, it may be convenient to use an *unnamed structure* instead of defining an explicit data structure to store the data, and then specifying its identifier. Unnamed structures are available for types scalar, variate, text, pointer, expression and formula. As you have seen in Section 1.1.1, an *unnamed scalar* may simply be a number. The other forms all have a common style: they start with an exclamation mark, then a *type code*, and then a list enclosed in round brackets. The type codes are:

(a) `V` (or `v`) for an *unnamed variate*. For example,

```
!V(164.2,160.6,163,166,159.8,163.9,161,161.3,165.8)
```

is an unnamed variate containing the zinc measurements in Example 1.1.2, and the statement

```
CALCULATE Zmed = MEDIAN( \
  !V(164.2,160.6,163,166,159.8,163.9,161,161.3,165.8) )
```

would calculate their median. If you do not specify any type code, `V` is assumed by default. So this example is the same as

```
!(164.2,160.6,163,166,159.8,163.9,161,161.3,165.8)
```

(b) `S` (or `s`) provides another way of specifying an unnamed scalar; this is likely to be useful mainly when defining procedures (5.3).

(c) `T` (or `t`) for an *unnamed text*. (Each value of a text is a string: see 2.3.2). For example,

```
!T(apples,pears)
```

is an unnamed text containing two strings: `apples` and `pears`. For a text containing a single string, an alternative is to give just the string within quotes. For example:

```
'apples'
```

(d) `P` (or `p`) for an *unnamed pointer* (2.6), when the list is of identifiers. For example,

```
!P(N,M,Q)
```

is a pointer containing the identifiers `N`, `M` and `Q`.

(e) `E` (or `e`) for an *unnamed expression*.

(f) `F` (or `f`) for an *unnamed formula*.

These last two are explained in 1.6.

1.4.4 System words

A *system word* is the name of a directive, or an option, or a parameter, or a function (1.6.1). The first character is a letter; subsequent characters are letters or digits. For example,

```
PRINT    print    Log    Log10
```

You can use capital and small letters interchangeably: thus the first two system words here are equivalent. System words can always be abbreviated to four letters; option and parameter names can often be abbreviated more than that (see 1.7.1). If more than four characters are supplied, Genstat checks the first 32 characters, but ignores characters 33 onwards. However, Genstat has few (if any) system words longer than 32 characters!

As for identifiers, if the `SET` directive (5.6.1) has been used to request *short wordlengths*, the ninth and subsequent characters of system words are ignored.

1.4.5 Missing values

A *missing value* indicates unknown information, and is represented by a single asterisk (`*`). When reading or printing data, missing values are represented by asterisks by default, but other representations can be used if you prefer (3.1.2 and 3.2.1).

1.4.6 Operators

An *operator* represents an arithmetic or logical operation, or some relationship between other kinds of item. Some operators have different meanings according to whether they appear in expressions or in formulae (1.6). Here is a list of all the operators and their names: more details are given in 4.1.1, 2:3.3.1 and 2:4.1.1.

Arithmetic operators	
addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`
exponentiation	`**`
matrix product	`*+`
Assignment operator	
assignment	`=`
Relational operators	
equality	`.EQ.` or `==`
string equality	`.EQS.`
non-equality	`.NE.` or `/=` or `<>`
string non-equality	`.NES.`
less than	`.LT.` or `<`
less than or equals	`.LE.` or `<=`
greater than	`.GT.` or `>`
greater than or equals	`.GE.` or `>=`
identifier equivalence	`.IS.`
identifier non-equivalence	`.ISNT.`
inclusion	`.IN.`
non-inclusion	`.NI.`
Logical operators	
negation	`.NOT.`
conjunction	`.AND.`
disjunction	`.OR.`
exclusive disjunction	`.EOR.`
Formula operators	
summation	`+`
dot product	`.`
cross product	`*`
nested product	`/`

deletion	`-`
crossed deletion	`-*`
nested deletion	`-/`
linkage of pseudo terms	`//`

Upper-case and lower-case letters can be used interchangeably for relational and logical operators. However the characters making up any of these operators must be contiguous; thus, for example, there must be no spaces between the dots and the letters of a relational operator.

1.5 Lists

A *list* is a set of items that are to be treated in the same way in a statement. The items are usually separated by commas (but not always: see 1.5.4).

Here are some examples of the three kinds of list:

```
VARIATE [VALUES=18,28,27,19,21] IDENTIFIER=Temp
TEXT [VALUES='London','Madrid','New York','Ottawa',\
  'Paris'] IDENTIFIER=City
PRINT STRUCTURE=City,Temp
```

The first set of values constitutes a *number list*, the second set a *string list*, and the `STRUCTURE` list for `PRINT` is an *identifier list.*

Missing values can occur in any of these lists. Their meanings and the ways of indicating them are described in 1.5.1, 1.5.2 and 1.5.3.

1.5.1 Number lists

Number lists appear in statements when values are put into a numerical data structure. Each item in a number list must be a number (1.4.1), a missing value, or the identifier of a scalar data structure; if an identifier, it stands for the value currently stored there. Missing values are interpreted as unknown observations in all directives that deal with numbers.

When numbers are to be listed in a repetitive or patterned series, you can save space and effort by compacting the lists as described in 1.5.4. Moreover, a set of numbers that form an arithmetic progression within a list can be written compactly using an *ellipsis*: this is three contiguous dots (`...`). For example,

```
1,2...10                  means                  1,2,3,4,5,6,7,8,9,10
```

In general, if *k*, *m* and *n* are numbers and *d* (=*m*−*k*) is the difference between *m* and *k*, then *k*,*m*...*n* stands for *k*, *k*+*d*, *k*+2*d*, *k*+3*d*, up to *n*. If *n* is not in the progression defined by *k* and *m*, then the progression ends at the value beyond which *n* would be passed (and this can sometimes be *k* itself). Here are two more examples:

```
-2, -1.5 ... 0.4          means                  -2, -1.5, -1, -0.5, 0
-2, -1.5 ... -1.6         means                  -2
```

If the last value in the progression is close to *n*, but not quite equal, this may be due to rounding error on the computer. For this reason, the last value of the progression will then be set to *n* itself; the precise criterion is to check if the last value is within *d*/100 of *n*.

When the step length *d* is plus or minus 1, you can compact the list even further. For example,

```
10...1                    is the same as         10,9...1
```

In general, the construction (*k*...*n*) is the same as *k*,*m*...*n*, where *m* is *k*+1 or *k*−1 depending on whether *n* is greater or less than *k*. If *k* equals *n*, the construction gives the single number *k*. You can leave out the brackets so long as there is no number preceding *k* that is not itself preceded by an ellipsis. For example,

```
1...3,5...8          means          1,2,3,5,6,7,8
1,2,3,5...8          means          1,2,3,5,7
1,2,3,(5...8)        means          1,2,3,5,6,7,8
```

1.5.2 String lists

String lists appear in two places. They occur when values are assigned to a structure that is to store text (as opposed to numbers), and they occur when the setting of an option or parameter is one or more string "tokens" that can be chosen from a specific list that has been defined for the directive. The second of these uses is described in 1.7.3.

For the first, each item in the string list must be a string (1.4.2), or a missing value. The latter is equivalent to the empty string `''`. An example of a string list with six items is

```
purple,'black and white','blue-green',so_dark,'2bright',F16
```

You can compact repetitive strings similarly to numbers (1.5.4).

Provided you have not used the `SET` directive (5.6.1) to request that newlines be ignored, the intermediate quotes and commas in a list of quoted strings can be replaced by newlines. For example

```
'Jack & Jill\
 went up the hill
To fetch a pail of water.'
```

is the same as

```
'Jack & Jill went up the hill','To fetch a pail of water.'
```

If the continuation symbol (\) were omitted, you would obtain

```
Jack & Jill',' went up the hill','To fetch a pail of water.'
```

while if you had previously specified `SET [NEWLINE=ignored]` (5.6.1), this would give the single string:

```
'Jack & Jill went up the hillTo fetch a pail of water.'
```

1.5.3 Identifier lists

Identifier lists are needed by many options and parameters of statements; they name the structures that are to be operated on. Each item in an identifier list must be an identifier, a qualified identifier (or qualified identifier list), a missing value (representing an unset item) or an unnamed structure (1.4.3). In addition, Genstat allows you to supply an *expression* (1.6.2) as the setting of any option or parameter that requires an identifier list; the calculation specified by the expression is evaluated to provide a list of results that are then used as the option or parameter setting. For example, we could put

```
PRINT LOG(Zinc)
```

to print the logarithms of the zinc measurements. (It is only sensible to do this though if the option or parameter is supplying *input* to the command!)

Qualified identifiers may occur in a list of identifiers to define subsets of the values of a data structure (i.e. substructures). The form is "*identifier* `$` *qualifier*", where the *qualifier* is a sequence of identifier lists enclosed in square brackets. For factors, variates, and texts, the qualifier has a single list, each element of which defines a subset of the vector concerned. For matrices there are two lists running in parallel, one for each dimension. For a symmetric matrix, there can be either one or two lists, depending on whether or not its two dimensions are to be subset in the same way. For a diagonal matrix there is a single list. Tables cannot be qualified. The elements of the qualifier lists can be scalars, numbers, variates, quoted strings, or texts. The set of units defined by an element in the qualification list is built up, by taking its values one at a time. Positive numbers (or texts or strings) add units to the set, while negative numbers delete the corresponding units from the set (if already there). A missing value can be used to include all the units, and one of these will be included implicitly at the start of the qualification list if the first element of the list is negative.

For example

```
V$[3,4]
```

means take the third and fourth items of data from the variate `V`. This is the same as

```
V$[3],V$[4]
```

which is a list of two scalars. If the structure has textual labels (2.3.2), these can be used in the qualification. For example

```
V$['c','d']
```

In general, a qualified identifier is

identifier `$ [`sequence of identifier lists`]`

If there are two lists, their elements are taken in parallel: for example,

```
M$[1,2; 4,5]
```

refers to a matrix `M`, and selects two elements: column 4 of row 1 and column 5 of row 2. So this is the same as

```
M$[1;4],M$[2;5]
```

A *qualified identifier* list can provide a more compact way of specifying a list of qualified identifiers. For example,

```
(A,B)$[1,2]
```

is the same as

```
A$[1],A$[2],B$[1],B$[2]
```

The general form is

`(`identifier list`) $ [`sequence of identifier lists`]`

If any list in the sequence is shorter than the others, it is repeated as often as is necessary; so for example

```
M$[1,2; 4,5,6]
```

is the same as

```
M$[1;4],M$[2;5],M$[1;6]
```

For further examples, see Section 4.1.6.

You can often compact a list of identifiers with suffixes by using a *suffix list*. For example,

```
A[1,2]
```

is the same as

```
A[1],A[2]
```

Identifier lists and suffix lists can be combined. For example,

```
(A,B)[1,2]
```

is the same as

```
A[1], A[2], B[1], B[2]
```

The lists are matched in lexicographic order: the items in the second list are matched in turn with the first item of the first list, then they are matched with the second item of the first list, and so on.

The empty suffix list `[]` stands for all suffixes of the identifiers preceding it. For example, if `P[1]`, `P[2]` and `P[3]` are the only current suffixed identifiers involving `P`, then `P[]` is the same as `P[1,2,3]`. Further examples are described in 2.6.

1.5.4 Ways of compacting lists

All three types of lists can be compacted by any of the methods described below, in addition to the methods described individually for each type of list earlier in this section.

The values of a structure can be substituted into a list using the substitution symbol (`#`). If `I` is an identifier, then `#I` is a list whose items are the values of the structure identified by `I`. If `I` is a pointer, then any item in `#I` that is itself a pointer is replaced by the values of that pointer (2.6).

For example, suppose `I` is a variate holding values `3,4`. Then

```
    1,2,#I
```

is the same as

```
    1,2,3,4
```

If `J` is another variate with values `1,2` then the same list could be written as

```
    #J,#I
```

Notice that this list is quite different from the list of two identifiers

```
    J,I
```

You can do the same with lists of strings. For example, if `Letters` is a text containing the strings `'b','c','d'`, then

```
    'a',#Letters
```

is the same as

```
    'a','b','c','d'
```

If you put a dummy (2.2.2) in a list, it is automatically replaced by the identifier that it is currently storing. So if the identifier of the dummy is preceded by `#`, then the values put into the list are those of the structure that the dummy is storing.

When a list is to contain a set of items repeated several times, you can use a *multiplier*. A multiplier is a number without a sign; it can also be #identifier, where the identifier is of a structure storing one non-negative number. A pre-multiplier repeats each item in the list in turn. For example,

```
    2(A,B,C)
```

is the same as

```
    A,A,B,B,C,C
```

A post-multiplier repeats the whole list:

```
    ('a','b')2
```

is the same as

```
    'a','b','a','b'
```

These can be combined. Thus,

```
    2(1...3)3
```

is the same as

```
    1,1,2,2,3,3,1,1,2,2,3,3,1,1,2,2,3,3
```

If the multiplier has the value 0, the construction contributes no items to the list. A multiplier with value 1 can be left out to give the form

```
    (list)
```

(You might want to use such a matched pair of brackets to indicate some grouping of items to anyone reading the program.)

You can compact a list of qualified identifiers by specifying a *qualified identifier* list:

```
(A,B)$[1,2]
```

is the same as

```
A$[1],A$[2],B$[1],B$[2]
```

The general form is

(identifier list) $ [sequence of identifier lists]

If any list in the sequence is shorter than the others, it is repeated as often as is necessary; so for example

```
M$[1,2; 4,5,6]
```

is the same as

```
M$[1;4],M$[2;5],M$[1;6]
```

1.6 Expressions and formulae

Expressions contain arithmetic and logical operations. They are required in commands that perform calculations. For example

```
CALCULATE Boundary = 2*(Width+Height)
```

An expression can also be given anywhere that a command expects a list of identifiers (1.5.3).

Formulae define the structure of a model in directives for some kinds of statistical analysis. For example

```
TREATMENTSTRUCTURE Drug*Rate
```

Both these constructions may contain *functions*. Details of functions are given in 4.2 for expressions and in 2:3.4 and 2:4.5 for formulae.

1.6.1 Functions

Functions have the form

function name (sequence of arguments)

A function name is a system word of one of the standard functions (4.2, 2:3.4 and 2:4.5). For example:

```
SQRT(X)    SUM(Y + 4*Z)
```

The function name can be abbreviated to four characters. If you give further characters they must match the full form up to the 32nd; characters beyond the 32nd are ignored (but few, if any, functions have names that long). However, if the `SET` directive has been used to request *short wordlengths*, the ninth and subsequent characters of the function name are ignored.

The *arguments* of a function are either lists or expressions; if there are several arguments they are separated by semicolons. For example:

```
CHISQ(2.5; 6)
```

1.6.2 Expressions

An *expression* consists of identifier lists, operators (1.4.6) and functions. Identifier lists must not include any missing identifiers, and the operators

```
.   -*   -/   //
```

cannot be used in expressions.

The simplest form of expression is an identifier list by itself, or a function by itself. You build up expressions from identifiers and functions by mixtures of three rules. Let `E` and `F` be expressions. Then these are also expressions:

```
(E)
```

monadic operator `E`

`E` *dyadic operator* `F`

The first means that putting brackets round an expression makes another expression. For the second, a *monadic operator* is an operator that works on just a single item: an example is minus in `-1`. In Genstat there are two monadic operators:

`.NOT.`	which negates a logical expression, and
`-`	which changes the sign of a numerical expression.

All the other operators work on pair of items: that this, they are *dyadic*. These operators (including the use of minus to mean subtract) can be used with the third rule. Other examples of expressions, illustrating the rules, are

```
5,6
A,B = -(C)
SUM(X) .EQ. 4
A = (B = C + 1) + 1
```

The precedence rules of the operators are very similar (but possibly not identical) to those in computer languages and programs like C, Fortran or Excel. The list below shows the precedence of all the operators in expressions when brackets are not used to make the order of evaluation explicit:

1) `.NOT.` Monadic `-`
2) `.IS.` `.ISNT.` `.IN.` `.NI.` `*+`
3) `**`
4) `*` `/`
5) `+` Dyadic `-`
6) `<` `>` `==` `<=` `>=` `/=` `<>` `.LT.` `.GT.` `.EQ.` `.LE.` `.GE.` `.NE.` `.NES.`
7) `.AND.` `.OR.` `.EOR.`
8) `=`

Within each class, operations are done from left to right within an expression. For example,

```
A > B+C/D*E
```

is the same as

```
A > ( B + ( (C/D) * E )
```

An identifier list in an expression can contain *qualified identifiers*; these select subsets of values of the structures (4.1.6). For example

```
V$[3,4]
```

means take the third and fourth items of data from the variate V. This is the same as

```
V$[3],V$[4]
```

which is a list of two scalars. If the structure has textual labels (2.3.2), these can be used in the qualification. For example

```
V$['c','d']
```

In general, a qualified identifier is

identifier `$ [`sequence of identifier lists`]`

If there are two lists, their elements are taken in parallel: for example,

```
M$[1,2; 4,5]
```

refers to a matrix `M`, and selects two elements: column 4 of row 1 and column 5 of row 2. So this is the same as

```
M$[1;4],M$[2;5]
```

You can compact a list of qualified identifiers in an expression by specifying a *qualified identifier* list:

```
(A,B)$[1,2]
```

is the same as

```
A$[1],A$[2],B$[1],B$[2]
```

The general form is

(identifier list) `$` [sequence of identifier lists]

If any list in the sequence is shorter than the others, it is repeated as often as is necessary; so for example

```
M$[1,2; 4,5,6]
```

is the same as

```
M$[1;4],M$[2;5],M$[1;6]
```

1.6.3 Formulae

A *formula* defines a statistical model; it consists of identifier lists, operators and functions. The identifier lists must not include any missing identifiers, and only the operators

```
+   -   *   /   .   -/   -*   //
```

can be used. The simplest form of a formula is an identifier list; also a formula can be a function by itself.

You build up other formulae by mixtures of two rules: if `M` and `N` are formulae, then so are

`(M)`

`M` operator `N`

For example

```
Sex * Diet
(Group / Variety) * Fertilizer
Drug * POL(Dose; 2)
```

The operators in a formula have the following precedence:

(1) `.`

(2) `//`

(3) `/`

(4) `*`

(5) `+   -   -/   -*`

Within each class, operations are done from left to right within a formula.

A formula is expanded into a series of *model terms*, linked by the summation operator (+). A model term contains one or more elements, separated from each other by the operator dot (`.`), each element being either an identifier or a function whose arguments are single identifiers. For example, the expanded form of the first formula above is

```
Sex + Diet + Sex.Diet
```

The interpretation of the terms is described in 2:3.3.1 and 2:4.1.1.

Identifiers in a list within a formula are treated as if they were separated by the summation operator and enclosed within brackets. For example

```
A,B * C
```

is the same as

```
(A + B) * C
```

The following table shows how operators combine terms, using `L` and `M` to represent two sums of terms.

Construction	**Expansion**
`L.M`	Sum of all pairwise combinations of terms in `L` with terms in `M` using the dot operator, with the terms ordered as explained

	below. For example: `(A+B).(C+D.E)` is the same as `A.C+B.C+A.D.E+B.D.E`
`L*M`	`L+M+L.M` ordered as explained below. For example `(A+B)*C` is the same as `A+B+C+A.C+B.C`
`L/M`	`L + L.M` where *`L`* is a term formed by combining all terms in `L` with the dot operator, ordered as explained below. For example `(A+B)/(C+D.E)` is the same as `A+B+A.B.C+A.B.D.E`
`L-M`	`L` without any terms that appear in `M`. For example `(A+B)-(A+C)` is the same as `B`
`L-/M`	`L` without any terms that consist of a term appearing in `M` combined with any other identifiers. For example `(A+B+B.C)-/B` is the same as `A+B`
`L-*M`	`L-M-/M` For example `(A+B+B.C)-*B` is the same as `A`

After expansions for the dot, slash and star operators, the terms are rearranged in order of increasing numbers of identifiers. Terms with the same number of identifiers are arranged in lexicographical order with respect to the order in which the identifiers first occurred in the formula itself.

A list of identifiers within a formula is treated as though the identifiers were linked by +, and contained within a pair of round brackets. For example

```
A * B,C
```

is equivalent to

```
A * (B + C)
```

and so it expands to

```
A + B + C + A.B + A.C
```

1.7 Statements

A *statement* is an instruction to Genstat, and has the general form:

statement-name `[`option-sequence`]` parameter-sequence terminator

For example,

```
READ [CHANNEL=2] STRUCTURE=Zinc,Chromium
```

If there are no options, the square brackets can be left out; but there must then be at least one space between the statement name and the first parameter setting: for example

```
PRINT STRUCTURE=Zinc; DECIMALS=2
```

Some directives have options but no parameters: for example,

```
SET [CASE=ignored]
```

makes upper-case and lower-case letters equivalent in identifiers (5.6.1). Others have neither options nor parameters. For example:

```
STOP
```

The statement name is one of three things: the name of one of the standard Genstat directives, or the name of a *procedure* (1.8.1 and 5.3), or the repetition symbol (&) described in 1.7.4.

The name of a directive can always be abbreviated to four characters. If more than four characters are given, the name is checked up to the 32nd character, but characters 33 onwards are ignored. (Note, though, that if the `SET` directive has been used to request *short wordlengths*, the ninth and subsequent characters of the function name are ignored.) The case of the letters (small or capital) is also ignored.

Names of procedures can also be abbreviated to four letters, provided there is no ambiguity with the names of directives or other procedures. Directives and procedures in the official Genstat library all have names that are distinct within the first four characters, so there should be no problem unless you (or your site) have defined procedures with ambiguous names. (Note, however, that some of the standard library procedures had to be renamed for Release 4.2 to avoid clashes with directive names: for example the original name of the procedure `FITNONNEGATIVE` differed from that of the directive `FITNONLINEAR` only in the seventh and eighth characters. The original names have been kept as synonyms but, for safety, should not be abbreviated beyond eight characters.)

Section 5.3 explains how you can write your own procedures, or attach *libraries* of stored procedures. Indeed, your site representatives can arrange to have a *site library* attached to Genstat automatically, just like the standard library. If the name of the command is ambiguous, Genstat selects the directive or procedure to use according to the following order of priority: directives, user-defined procedures, procedures in libraries attached by the user (in order of channel number), procedures in the site library, and procedures in the official library. Again, if more than four characters are specified, Genstat checks the name only up to the 32nd character (or the eighth if short wordlengths have been requested), and the case of the letters is ignored.

The terminator of a statement is colon (`:`). Thus the line

```
VARIATE [NVALUES=12] Sales : READ Sales
```

contains two statements.

Alternatively, you can usually end a statement by pressing the carriage-return key (`<RETURN>`). In other words, newline is normally synonymous with colon. You can change this with the `SET` directive (5.6.1):

```
SET [NEWLINE=ignored]
```

indicates that newlines are to be ignored in the rest of the program.

Even if newlines are not ignored, there are still three situations when a newline will not end a statement.

(a) When newline occurs within a string, it terminates that string and begins another (1.5.2).

(b) A newline within a comment is ignored (along with the rest of the comment): for example

```
PRINT STRUCTURE=Zmin,Zmean,Zmax; FIELDWIDTH=8,9,8;"
"DECIMALS=1,2,1
```

is a single statement.

(c) You can indicate that a statement is to continue onto the next line by putting a continuation symbol (\) before pressing `<RETURN>` for example,

```
PRINT STRUCTURE=Zmin,Zmean,Zmax; FIELDWIDTH=8,9,8;\
DECIMALS=1,2,1
```

is again a single statement. Any characters between the continuation symbol and the end of the line are ignored. Genstat does however have the limitation that a statement must not exceed 2048 characters, after deletion of extraneous spaces.

1.7.1 Syntax of options and parameters

The sequences of options and parameters specify the items upon which the statement is to operate: these items are called the *arguments* of the statement. A sequence consists of one or more settings, each separated from the next by a semicolon (`;`). You can see an example of a sequence of parameter settings in the `PRINT` statement above. Each setting, whether of an option or a parameter, has one of the general forms:

name = list

name = expression

name = formula

The list, expression or formula can be null (length zero). Rules by which the "name=" can be left out are defined below; the types of setting are discussed further in 1.7.3.

An *option name* is a system word, which can be abbreviated to the minimum number of letters needed to distinguish it from the options that precede it in the prescribed order for the directive or procedure concerned. Characters up to the 32nd (or the eighth if short wordlengths have been requested) must match the appropriate part of the full form; subsequent characters are ignored. For example, here are the options of the TABULATE directive (4.11.1), with the minimum form of each name printed in bold:

PRINT, **C**LASSIFICATION, **CO**UNTS, **S**EQUENTIAL, **M**ARGINS,
IPRINT, **W**EIGHTS, **PE**RCENTQUANTILES, **O**WN, **OWNF**ACTORS,
OWNVARIATES, **IN**CHANNEL, **INF**ILETYPE

Notice for example that the minimum for COUNTS is CO, since C on its own would not distinguish it from CLASSIFICATION which precedes it in this prescribed order.

A *parameter name* is also a system word, and has the same abbreviation rule as the option names. For example, the parameters for TABULATE are (with minimum forms again in bold):

DATA, **T**OTALS, **N**OBSERVATIONS, **M**EANS, **MI**NIMA, **MA**XIMA,
VARIANCES, **Q**UANTILES, **S**DS, **SK**EWNESS, **K**URTOSIS, **SE**MEANS,
SESKEWNESS, **SEK**URTOSIS

You usually need type no more than one or two characters for any option or parameter name; there are no directives or procedures in the standard library that require more than four characters for their option and parameter names. However, if you are likely to refer to a statement again in future (as for example if it is part of a procedure), remember that it may be difficult to understand if it is abbreviated too heavily.

You can omit the name and the equals character altogether by taking account of the prescribed order of options, or of parameters, within the directive or procedure. The rules for parameters are the same as those for options, and are as follows:
(a) If the first option setting in a statement is for the first option defined for that directive or procedure, then "name=" can be omitted.
(b) The "name=" can also be omitted for later option settings if the preceding setting is for the option immediately before that option in the prescribed order. For example,

```
TABULATE [PRINT=totals,means; COUNTS=Rep; SEQUENTIAL=Sval]\
  DATA=Spending; MEANS=Meansp
```

can be abbreviated to

```
TABULATE [totals,means; COUNTS=Rep; Sval] \
  Spending; MEANS=Meansp
```

You can omit "PRINT=" here by rule (a) as it is the first option in the order prescribed for TABULATE. Similarly, "DATA=" can be omitted as DATA is the first parameter. You can omit "SEQUENTIAL=" by rule (b), because COUNTS which precedes it here is also the option that precedes SEQUENTIAL in the definition of TABULATE. However, you cannot omit "COUNTS=", because in the prescribed order there is another option between COUNTS and PRINT. The same is true for "MEANS=".

An option or parameter setting can be *null*: that is, it can have a list of length zero, or a null expression, or a null formula. Thus, by putting a null setting for the CLASSIFICATION option and the TOTALS and NOBSERVATIONS parameters, all the names can be omitted:

```
TABULATE [totals,means; ; Repl; Sval] Spending; ; ; Meansp
```

If a directive has a single parameter, no name is defined. For example, there is no name for the expression that is the only parameter for the CALCULATE directive (4.1.1).

1.7.2 Roles of options and parameters

Parameters specify parallel series of arguments that are operated on in turn when the statement is carried out. For example, in

```
TABLE [CLASSIFICATION=Age,Sex] \
  IDENTIFIER=Income,Cars,Spending; DECIMALS=2,0,2
```

there are two parameters: `IDENTIFIER` and `DECIMALS`. The statement declares three tables and defines their default numbers of decimal places. So, when they are printed later in the program `Income` will have two decimal places, `Cars` will have none, and `Spending` will have two.

The main information in a directive or procedure is usually given by the first parameter, and so this is said to define the series of *primary arguments*. In the example, they are `Income`, `Cars` and `Spending`. Usually the parameter setting is an identifier list, and so the series is a list of data structures.

Alternatively, the setting of the first parameter can be an expression, in which case the first identifier list in the expression is the series of primary arguments. For example, if `A`, `B`, `M`, `N`, `P` and `Q` are variates, then in

```
CALCULATE A,B = M,N + P,Q
```

the primary arguments are `A` and `B`.

Another possibility is that the setting may be a formula, in which case the expanded list of model terms (1.7.3) is the series of primary arguments. For example, in the regression statement

```
FIT A * B
```

the primary arguments are the terms `A`, `B` and `A.B` (representing the main effects of A and B and their interaction, see 2:3.3.1),

Later parameters, or other lists within an expression, specify *secondary arguments* which run in parallel with the primary arguments, and provide ancillary information. Examples of secondary arguments are in the `TABLE` statement above, and on the right-hand side of the assignment operator in the `CALCULATE` statement.

The series of primary arguments should always be the longest; if a series of secondary arguments is longer, you are given a warning and elements beyond the length of the primary series are ignored. Any series that is shorter is recycled: that is, the series is traversed again, as many times as is necessary to match the length of the primary series. Thus the `TABLE` declaration above means exactly the same if it is written

```
TABLE [CLASSIFICATION=Age,Sex] \
  IDENTIFIER=Income,Cars,Spending; DECIMALS=2,0
```

Options, on the other hand, specify information that applies to all the primary arguments (with their corresponding secondary arguments). Thus in the `TABLE` example above, all three tables are classified by `Age` and `Sex`.

Many options have *default* values, namely values that are assumed if the option is not set explicitly in a statement. But some options have to be set, for example the `OLDSTRUCTURE` and `NEWSTRUCTURE` options of the `COMBINE` directive (4.11.4). Some parameters also have defaults: for example, the `METHOD` option of `LPGRAPH` (6.10.1) assumes point plots. In a very few directives, the primary parameter also has a default; for example, the `Y` parameter of `RKEEP` (2:3.1.4) assumes the list of current response variates.

1.7.3 Types of option and parameter settings

An option or parameter setting may need a formula, or an expression, or a list (1.7.1).

When the setting is an expression, and the parameter or option has a defined name, the name and its accompanying equals character cannot be left out if the expression begins with "unsuffixed identifier=". This is because there would then be confusion between the name of the option or parameter and the unsuffixed identifier. For example, you could not leave out the name `CONDITION` in the statement

```
RESTRICT STRUCTURE=Income; CONDITION=Agecond=Age>30
```

That is, if you wrote

```
RESTRICT Income; Agecond=Age>30
```

Genstat would try to interpret `Agecond` as a parameter name, and a message would be printed alerting you to this syntax error. You could, however, put the expression in brackets:

```
RESTRICT Income; (Agecond=Age>30)
```

No such problem arises with directives like `CALCULATE` (4.1.1), `CASE`, `IF` and `ELSIF` (5.2), because in these the expression is the only parameter, and thus has no defined name. For example, you can write

```
CALCULATE Agecond = Age>30
```

(And note also that there is no need for the condition expression in `RESTRICT` to include an assignment! See 4.4.1.)

Many options and parameters need lists of identifiers. Any restrictions on the types of identifiers for particular lists are mentioned along with the descriptions of the syntax in later chapters. For example, the specification of the `CLASSIFICATION` option of the `TABLE` directive (2.5) states

`CLASSIFICATION` = *factors*   Factors classifying the tables; default `*`

No structures other than factors can be used here.

The options or parameters that require lists of strings almost always select them from a list of *string tokens*, defined by Genstat for that option or parameter. (The one exception amongst the directives is the `VALUES` option of `TEXT`; see 2.3.2.) For example, the `PRINT` options of `ADISPLAY` (2:4.1.3) and `ANOVA` (2:4.1.2) have possible tokens:

aovtable, **i**nformation, **c**ovariates, **e**ffects, **r**esiduals,
contrasts, **m**eans, **cb**effects, **cbm**eans, **s**tratumvariances, %cv,
missingvalues

These let you choose which components of output are to be printed from an analysis of variance. The rules for string tokens are exactly the same as those for option and parameter names (1.7.1): they may be typed in capital or small letters (or mixtures), and each one can be abbreviated to the minimum number of characters necessary to distinguish it from earlier values in the list. If more than that number is given, the extra characters must match the full form up to the 32nd character (or the eighth if short wordlengths have been requested).

The minimum forms of the tokens for the `PRINT` options, above, are marked in bold. Thus

```
PRINT=Aovtable,Effects,MissingValues
```

is the same as

```
PRINT=aovtable,effects,missingvalues
```

and both of these can be abbreviated, for example, to

```
PRINT=a,e,mi
```

To prevent any printing at all, with the `PRINT` option of any directive, you specify a missing string:

```
PRINT=*
```

or

```
PRINT=''
```

The special symbol `#` provides a succinct way of specifying the default setting of any option. This is most useful in options like `PRINT` above, where you might want to ask for the default plus some extra output. The default of the `PRINT` option for `ANOVA` is `aovtable, information, covariates, means, missing`. So, to print the residuals as well, you can simply put

```
PRINT=#,residuals
```

which will have the same effect as

```
PRINT=aovtable,information,covariates,means,missing,\
      residuals
```

Number lists are needed by the `VALUES` options of the directives that define numerical data structures (2.1.1), but not by the options or parameters of any other directive.

1.7.4 Repetition of a statement and its options

You can repeat a directive name by typing the ampersand character (`&`). At the same time you can reset as many options as you want; those that you do not mention remain as in the previous statement. For example, after

```
READ [PRINT=data; CHANNEL=2] Costs
```

the statement

```
& Profits
```

is equivalent to

```
READ [PRINT=data; CHANNEL=2] Profits
```

while the statement

```
& [CHANNEL=3; REWIND=yes] Profits
```

is equivalent to

```
READ [PRINT=data; CHANNEL=3; REWIND=yes] Profits
```

You need not type a colon or newline before an ampersand, as it automatically terminates the previous statement.

1.8 Ways of compacting programs

You can store Genstat statements in two ways: in a procedure or in a macro.

1.8.1 Procedures

A *procedure* is a series of complete Genstat statements. It is like a procedure in Basic or Pascal, a subroutine in Fortran or a function in C or C++. These statements are self-contained, in that all the data structures that they use are accessible only within the procedure, apart from those explicitly defined as options or parameters of the procedure. Rules for writing and defining procedures are described in 5.3. The rules of syntax for using a procedure are identical to those for the standard Genstat directives (1.7); indeed, since you can get access to procedures automatically from libraries, you do not have to know whether a particular statement uses a directive or a procedure.

1.8.2 Macros

A *macro* is a Genstat text into which you have placed a section of Genstat program. The text must have an unsuffixed identifier. You can substitute the contents of the macro into the program by a contiguous pair of hash characters `##`; the substitution takes place immediately after Genstat reads the statement that contains the hash characters.

A simple kind of macro would be a part of a Genstat statement. For example,

```
TEXT [VALUES='[PRINT=data,summary; CHANNEL=2]'] Optset
```

assigns to a text with identifier `Optset` the string between the single quotes. If you later type

```
READ ##Optset Patient,Sex,Weight
READ ##Optset Calories,Wtgain
```

then `Optset` is treated as a macro and its contents are inserted into each of the two statements; so the named structures are read using the options for `PRINT` and `CHANNEL` defined in the string that has been put in `Optset`. Defining `Optset` in this way saves effort in typing the two `READ` statements; it would also allow you to change the options of both statements simultaneously.

More complicated macros may contain complete statements. For example, suppose that the computer file `Alg.dat` contains three lines, each a quoted string (1.4.2):

```
'CALCULATE Previous = Root'
'& Root = (X/Previous + Previous)/2'
'PRINT STRUCTURE=Root,Previous; DECIMALS=4' :
```

These three statements can be read into a text for use as a macro. A simple program for calculating the square root of 48 (without using the standard function `SQRT`) can then conveniently be written as follows:

```
SET [INPRINT=statements,macros]
SCALAR IDENTIFIER=X,Root; VALUE=48
TEXT [NVALUES=3] Estsqrt
OPEN NAME='Alg.dat'; CHANNEL=2
READ [CHANNEL=2] STRUCTURE=Estsqrt
##Estsqrt
##Estsqrt
##Estsqrt
PRINT [IPRINT=*] '3 iterations estimate square root of 48
as',Root
```

Output from running this program in batch is shown below.

Example 1.8.2

```
   2  SET [INPRINT=statements,macros]
   3  SCALAR IDENTIFIER=X,Root; VALUE=48
   4  TEXT [NVALUES=3] Estsqrt
   5  OPEN NAME='Alg.dat'; CHANNEL=2
   6  READ [CHANNEL=2] STRUCTURE=Estsqrt

    Identifier   Minimum      Mean   Maximum    Values   Missing
       Estsqrt                                       3         0

   7  ##Estsqrt
     1  CALCULATE Previous = Root
     2  & Root = (X/Previous + Previous)/2
     3  PRINT STRUCTURE=Root,Previous; DECIMALS=4

        Root    Previous
     24.5000     48.0000

   8  ##Estsqrt
     1  CALCULATE Previous = Root
     2  & Root = (X/Previous + Previous)/2
     3  PRINT STRUCTURE=Root,Previous; DECIMALS=4

        Root    Previous
     13.2296     24.5000

   9  ##Estsqrt
     1  CALCULATE Previous = Root
     2  & Root = (X/Previous + Previous)/2
     3  PRINT STRUCTURE=Root,Previous; DECIMALS=4

        Root    Previous
      8.4289     13.2296

  10  PRINT [IPRINT=*] '3 iterations calculate the square root of 48 as',Root

 3 iterations calculate the square root of 48 as       8.429
```

The first statement (in line 2) arranges to print statements and contents of macros. Then `X` and `Root` are defined as scalars, and both are given the value 48. `Estsqrt` is defined as a text with three values (or lines), and read from `Alg.dat` in lines 5 and 6. The first line is numbered 2 here (and in many of the other programs in the Guide) because Genstat *for Windows*, which was used to run this example, automatically generated an initial statement (not shown above) to set the "working directory" to the folder containing the file `Alg.dat`. You can choose the working directory using the Select Input File menu.

The macro is substituted into the program three times: because of `SET`, its contents are printed each time, with line numbers indented by two characters. The `IPRINT` option of `PRINT` in line 10 prevents printing of the identifier `Root`: all that appears is the number stored in `Root`. As you can see from the output, the value is still some way from convergence. Methods of testing for convergence in iterative algorithms like this are described in 5.2.4.

Substitution using `##` takes effect immediately after Genstat has read the relevant input line. For macros that contain complete statements, like `Estsqrt`, an alternative is to use the `EXECUTE` directive (5.4.3). The substitution will then take place only when the `EXECUTE` statement is executed. This makes no difference in ordinary programs, but is very useful inside procedures or loops, where the statements are defined before they are executed (5.3 and 5.2.1).

1.9 Conventions for examples in later chapters

You have now seen that you can lay out your programs in many ways. You can include spaces to make them more readable, or you can leave spaces out to make them compact (1.3.5). You can type statements spread over several lines, or you can have more than one to a line (1.7). You can write system words in capital letters, or in small letters, or in a mixture, and you can use the full or the abbreviated forms (1.4.4 and 1.7). You can do the same with strings in options and parameters (1.7.3). You can write identifiers with capital letters or with small letters, or in a mixture, and you can control whether or not these are equivalent (1.4.3).

In this Guide, however, we have imposed some conventions. The use of spaces is standardized. System words are given in full and in capitals; the only exception is that the name, and corresponding equals character, of the main parameter of a directive will usually be left out in later chapters. String tokens are given in full and in small letters. Identifiers will begin with a capital; any other letters are small. There is usually only one statement per line, unless this is very wasteful of space; continuation lines are indented.

We hope these conventions will help you to recognize the items, both in the descriptions of syntax and in the examples. However, in your own programs, you should develop your own style according to what you find most convenient.

2 Data structures

Data structures store the information on which a Genstat program operates. Examples include data for statistical analyses, coordinates for graphs, text for annotation, and so on. You can also store almost anything that can be printed in an analysis. This enables you to extend the range of facilities that Genstat offers, by taking information from one directive and using it as input for another. To allow you to do this, Genstat has a comprehensive set of different structures. However, there are many similarities across the directives that are used to define them, and the more complicated structures are required only for the more advanced uses of Genstat. You can define the identifier of a structure, together with its type, using a directive known as a *declaration*. The directive for declaring each type of structure has the same name as given to that type of structure, for example `SCALAR` to declare a scalar (or single-valued numerical structure), and so on. These are the directives, with details of their corresponding data structures and references to the sections where they are described below.

`SCALAR`	single number (2.2.1)
`VARIATE`	series of numbers (2.3.1)
`TEXT`	series of character strings i.e. lines of text (2.3.2)
`FACTOR`	series of group allocations, represented by a pre-defined set of numbers or strings (2.3.3)
`MATRIX`	rectangular matrix (2.4.1)
`DIAGONALMATRIX`	diagonal matrix (2.4.2)
`SYMMETRICMATRIX`	symmetric matrix (2.4.3)
`TABLE`	table − to store tabular summaries like means, totals etc (2.5)
`DUMMY`	single identifier (2.2.2)
`POINTER`	series of identifiers e.g. to represent a set of structures (2.6)
`EXPRESSION`	arithmetic expression (2.2.3)
`FORMULA`	model formula − to be fitted in a statistical analysis (2.2.4)
`LRV`	latent roots and vectors (2.7.1)
`SSPM`	sums of squares and products with associated information such as means (2.7.2)
`TSM`	model for Box-Jenkins modelling of time series (2.7.3)
`TREE`	tree, as used to represent classification trees, identification keys and regression trees (2.8, 2:6.20, 2:6.21, 2:3.9)

You can also define data structures whose contents are customized for particular tasks (2.7.4).

`STRUCTURE`	defines a customized data structure
`DECLARE`	declares one or more customized data structures

In the standard version of Genstat, your program can contain as many data structures of each type as you like, limited only by the total amount of workspace that they occupy. Student Versions may have additional constraints, explained in the accompanying on-line help or documentation.

This chapter also describes several additional commands that are useful for managing your data structures.

`DELETE`	allows values of data structures to be deleted to save space within Genstat; attributes can also be deleted so that the structure can be redefined, for example as another type (2.10.1)
`RENAME`	renames a data structure, to give it a new identifier

	(2.10.2)
`DUPLICATE`	forms new data structures with attributes taken from an existing structure (2.10.3)
`PDUPLICATE`	duplicates a pointer, with all its components (2.10.4)
`LIST`	lists the data structures currently in store (2.11.1); in Genstat *for Windows* an alternative is to use the Data Display window obtained by pressing F5
`DUMP`	prints attributes and values of data structures (2.11.2)
`GETATTRIBUTE`	accesses attributes of data structures such as their types, sizes and so on (2.11.3)

2.1 Declarations

Most data structures have a name; the exceptions are called *unnamed structures* and are described in 1.4.3. The name is called an *identifier*, and this is used to refer to the structure within your program. As explained in (1.4.3), an ordinary identifier starts with a letter (1.3.1) and then contains digits (1.3.2) or letters (or both). Note, however, that some of the menus in Genstat *for Windows* set up private temporary structures with identifiers that begin with the underscore character _ (which, within Genstat, is treated as a letter). So, you may find it safest to avoid starting your own identifiers in this way.

Genstat stores only the first 32 characters; subsequent characters are ignored. (This is the default action, but you can used the `SET` directive (5.6.1) to request that only eight characters are stored, as in releases earlier than Release 4.2.) Identifiers can also have suffixes, enclosed in square brackets; further details are given in 1.4.3 and 2.6.

You can define the identifier of a structure, together with its type, using a directive known as a *declaration*. There is a directive available to declare each type of structure. For example the declaration

```
SCALAR Length
```

uses the `SCALAR` directive to define a scalar with identifier `Length`.

You can declare several structures in a single statement: for example

```
SCALAR Length,Width,Height
```

declares `Length`, `Width` and `Height` all to be scalars. A declaration need define only the identifier and the type. However, you can also specify values for the data structures (2.1.1), as well as various attributes that carry ancillary information about the structures.

As well as the identifier, most data structures can also be given an "extra" text of descriptive information that is displayed as well as the identifier in output from many Genstat commands (2.1.3). You can also request that the "extra" text is used in the output instead of the identifier. So, as far as the output is concerned, there need be no restrictions on the name that you use for a data structure.

Some attributes must be specified before the structure can be given values, for example the number of rows and columns of a matrix (2.4.1). Others need be set only if you choose to use them; for example, the number of decimal places (2.1.2) to be used by default when printing the values.

Options and parameters that apply generally to several different directives are described in this section; the others are described with the directive concerned, later in the chapter.

2.1.1 The `VALUES` option and parameter

Most declarations have an option and a parameter for specifying values for the structures that are defined. The same name is used for both purposes: it is `VALUE` if the structures are of a type that stores a single value, and `VALUES` if they can each store several. The option defines a common value (or set of values) for all the structures in the declaration, while the parameter allows the structures each to be given different values.

With the option you must supply a list of values. With the parameter, however, you must give a list of identifiers of data structures of the appropriate mode; the unnamed structures described in 1.4.3 are particularly useful for this. Thus, to declare variates X and Xsq each with its own set of values, you can put:

```
VARIATE X,Xsq; VALUES=!(1,2,3,4),!(1,4,9,16)
```

X then contains the values 1 up to 4, and Xsq contains 1, 4, 9 and 16.

If both the option and the parameter are specified, the parameter takes precedence. So

```
SCALAR [VALUE=12.5] Length,Width,Height; VALUE=*,*,200
```

gives Length and Width the value 12.5 and Height the value 200. (The asterisk in the identifier list for the VALUE parameter means an omitted entry: see 1.5.3.)

2.1.2 The DECIMALS parameter

In the declarations of structures that contain numbers, there is a DECIMALS parameter for defining the number of decimal places that Genstat will use by default whenever the values of each structure is printed. This applies to output either by PRINT or from an analysis (but it does not affect the accuracy with which the numbers are stored). For example,

```
SCALAR Length,Width,Height; VALUE=12.5,6.25,120;\
  DECIMALS=1,2,0
```

specifies that Length, Width and Height should in future be printed with one, two and zero decimal places respectively, although you can of course override this within the PRINT directive itself (3.2.1 and 3.2.2).

Procedure DECIMALS can be used to set the DECIMALS parameter automatically to the minimum number of decimal places to print the structure exactly. For example

```
DECIMALS Length,Width,Height
```

Also, the SET directive has an option SIGNIFICANTFIGURES which allows you to control the precision to be used when DECIMALS has not been set either in PRINT or when the structures concerned were declared (5.6.1).

2.1.3 The EXTRA parameter and IPRINT option

You can associate a text with each data structure by means of the parameter EXTRA. This text may then be used to give a fuller annotation of output. For example:

```
SCALAR Length,Weight; EXTRA=' in centimetres',' in grams'
```

The IPRINT option allows you to control when the extra text is used in output. If IPRINT is not set, the data structure will be identified in whatever way is usual for the section of output concerned. For example, the PRINT directive generally uses the identifier (although this can be changed using the IPRINT option of PRINT itself), while the ANOVA directive prints both the identifier and the extra text for a y-variate. If you set IPRINT=identifier, only the identifier will be used. Alternatively, if you set IPRINT=extra, only the extra text will be used (so you can then label the data structure in any way that you want). Finally, if you set IPRINT=extra,identifier, both the identifier and the extra text will be used.

2.1.4 The MINIMUM and MAXIMUM parameters

These two parameters allow you to define lower and upper limits on the values expected for any structure that stores numbers. Genstat then prints warnings if any values outside that range are assigned to the structure.

2.1.5 The `DREPRESENTATION` parameter

Dates and times can be stored in any numerical Genstat data structure. The time is contained in the decimal part of the number, and represents the time during the day. So, for example, 12 noon is stored as 0.5, and 6 a.m. as 0.25. The date is represented in the integer part of the number, and contains the number of days since the *base date* of 29th February 1600 (see 1.4.1). The `DREPRESENTATION` parameter can be used to define a default format to be defined for use when the data structure concerned is printed. The setting of the parameter is either a scalar indicating a predefined format, or a string defining a custom format.

The string for a custom format contains a sequence of keys to represent the required components of the date and time. The available keys are:

`d`	day number within the month, using the minimum number of digits (e.g. 3, 12)
`dd`	day number within the month, using two digits (e.g. 03, 12)
`dth`	day number with one digit and suffix (e.g. 3rd, 12th)
`m`	month number, using the minimum number of digits
`mm`	month number, using two digits
`mmm`	abbreviated month name (Jan, Feb, Mar, Apr, May, June, July, Aug, Sept, Oct, Nov, Dec)
`mmmm`	month name in full
`yy`	year as a two-digit number (omitting the century)
`yyyy`	year as a four-digit number (including the century)
`weekday`	day of the week (Monday, Tuesday, and so on)
`wday`	abbreviated day of the week (Mon, Tue, Wed, Thur, Fri, Sat, Sun)
`time24`	time, including seconds, using a 24 hour clock
`time12`	time, including seconds, using a 12 hour clock (with a.m. and p.m.)
`time100`	time, using 24 hour clock and including hundredths of seconds
`hours`	elapsed time in hours, minutes and seconds
`hours100`	elapsed time in hours, minutes, seconds and hundredths of seconds
`minutes`	elapsed time in minutes and seconds
`minutes100`	elapsed time in minutes, seconds and hundredths of seconds
`seconds`	elapsed time in seconds
`seconds100`	elapsed time in seconds and hundredths of second

You can also insert one or more separators between the keys, any combination of space ( ), slash (`/`), hyphen (`-`) and comma (`,`).

Note: the operation of the 2-digit representation of a year is controlled by a "break point". This has the initial default of 30, but that can be changed by the `YEAR2DIGITBREAK` option of `SET`, or in the `DateFormat` tab of the `Options` menu in Genstat *for Windows*. With the initial default of 30, for example, years from 1930 to 2029 will be represented as two digits, but others will be printed with four digits.

To simplify the specification of the most commonly used formats, a range of standard pre-defined formats are available. These are specified by supplying a scalar containing one of the numerical codes in the left-hand column of the table below.

code	**format**	**example**
1	dd/mm/yy	03/08/98
2	dd/mm/yyyy	03/08/1998
3	d/m/yy	3/8/98

4	d/m/yyyy	3/8/1998
5	ddmmyy	030898
6	ddmmyyyy	03081998
7	ddmmmyy	03Aug98
8	ddmmmyyyy	03Aug1998
9	dd-mmm-yy	03-Aug-98
10	dd-mmm-yyyy	03-Aug-1998
11	dmmmyy	3Aug98
12	dmmmyyyy	3Aug1998
13	d-mmm-yy	3-Aug-98
14	d-mmm-yyyy	3-Aug-1998
15	d-mmmm-yy	3-August-98
16	d-mmmm-yyyy	3-August-1998
17	yymmdd	980803
18	yyyymmdd	19980803
19	yy/mm/dd	98/08/03
20	yyyy/mm/dd	1998/08/03
21	mmddyy	080398
22	mmddyyyy	08031998
23	mm/dd/yy	08/03/98
24	mm/dd/yyyy	08/03/1998
25	mmm-dd-yy	Aug-03-98
26	mmm-dd-yyyy	Aug-03-1998
27	yyyy-mm-dd	1998-08-03
28	weekday, dth mmmm yyyy	Monday, 3rd August 1998
29	weekday	Monday
30	mmm-yy	Aug-98
31	yy	98
32	yyyy	1998
33	dd-mmm-yyyy time100	03-Aug-1998 18:55:30.35
34	yyyy-mm-dd time (ODBC Standard format)	1998-08-03 18:55:30
35	dd-mmm-yyyy time12	03-Aug-1998 6:55:30 pm
36	time24	18:55:30
37	time12	6:55:30 pm
38	hours	48:55:30
39	seconds	68538.35
40	dd/mm/yyyy time24	03/08/1998 18:55:30
41	m-yy	8-98
42	m-yyyy	8-1998
43	mm-yy	08-98
44	mm-yyyy	08-1998
45	d/m	3/8
46	dd/mm	03/08
47	d-mmm	8-Aug
48	dd-mmm	08-Aug
49	mmm	Aug

You can also use the custom date formats supported by the client in Genstat *for Windows*. See the `Date Formats` page in the on-line help for details.

2.1.6 The `MODIFY` option

Normally if you declare a data structure for a second time, you will lose all its existing attributes and values. If you want to retain them you should set option `MODIFY=yes`. Thus, to redeclare the scalar `Length`, changing only its number of decimals to two, you would need to put

```
    SCALAR [MODIFY=yes] Length; DECIMALS=2
```

The one attribute that you cannot readily redefine is the type. Before you can redeclare an identifier to refer to a structure of a different type, you must delete all its attributes. (See Section 2.10, where there is an example redeclaring a variate as a text.)

2.2 Single-valued data structures

2.2.1 Scalars

A scalar data structure stores a single number (1.4.1). The `SCALAR` directive which declares scalars has only the general options and parameters already described in 2.1.

`SCALAR` directive

Declares one or more scalar data structures.

Options

`VALUE` = *scalar*	Value for all the scalars; default is a missing value
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used to identify the scalars in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the scalars
`VALUE` = *scalars*	Value for each scalar
`DECIMALS` = *scalars*	Number of decimal places for printing
`EXTRA` = *texts*	Extra text associated with each identifier
`MINIMUM` = *scalars*	Minimum value for the contents of each structure
`MAXIMUM` = *scalars*	Maximum value for the contents of each structure
`DREPRESENTATION` = *scalars* or *texts*	Default format to use when the contents represents a date and time

---

`SCALAR` is the one type of declaration where values are defined by default: if you do not define a value explicitly for a scalar, Genstat gives it a missing value.

Examples are given in 2.1. Unnamed scalars (which may just be simple numbers) are described in 1.4.3.

2.2.2 Dummies

A *dummy* is a data structure that itself stores the identifier of some other structure. You will find this useful in identifier lists, where in nearly all cases Genstat replaces a dummy by the identifier that it stores. The only exceptions are the `IDENTIFIER` parameter of the `DUMMY` directive itself (see below), the `STRUCTURE` parameter of `ASSIGN` (4.9.1), the parameters of `FOR`, and in the `SET` and `UNSET` functions in expressions (4.2.6).

Dummies are particularly useful when you want the same series of statements to be used with several different data structures. By using a dummy structure within the statements, you can make them apply to whichever structure you require. The dummy structure is like a plug which can be connected to the structure that you need; the important point is that you can then connect another structure without changing the statements themselves.

The most obvious situations where this is useful are in loops and procedures, and there the dummies are declared automatically as explained in 5.2.1 and 5.3.2.

To declare a dummy explicitly, you use the `DUMMY` directive. This has only the general options and parameters already described in 2.1.

`DUMMY` directive

Declares one or more dummy data structures.

Options

`VALUE` = *identifier*	Value for all the dummies; default `*`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`PRINT` = *string tokens*	Information to be used by default to identify the dummies in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the dummies
`VALUE` = *identifiers*	Value for each dummy
`EXTRA` = *texts*	Extra text associated with each identifier

For example:

```
DUMMY Xdum,Ydum; VALUE=Day,Growth
```

2.2.3 Expression data structures

The expression data structure stores a Genstat expression (1.7.2), for example

```
Hours = Minutes/60
```

Usually you will find it easiest to type out an expression like this explicitly whenever you need it. The main use, then, for this rather specialized data structure is to supply an expression as the argument of a procedure.

Options and parameters of the `EXPRESSION` directive, which declares expressions, are already described in 2.1.

`EXPRESSION` directive

Declares one or more expression data structures.

Options

`VALUE` = *expression*	Value for all the expressions; default `*`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used by default to identify the expressions in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the expressions
`VALUE` = *expression structures*	Expression data structures providing values for the expressions
`EXTRA` = *texts*	Extra texts associated with the identifiers

Here are two examples using the `VALUE` option:

```
EXPRESSION [VALUE=Length*Width*Height] Vcalc
EXPRESSION [VALUE=Dose=LOG10(Dose)] Dtrans
```

These put the expression `Length*Width*Height` into the identifier `Vcalc`, and the expression `Dose=LOG10(Dose)` into `Dtrans`. Both expressions could be declared simultaneously, using the `VALUE` parameter, by putting

```
EXPRESSION Vcalc,Dtrans; VALUE=!E(Length*Width*Height), \
  !E(Dose=LOG10(Dose))
```

Rules for omitting "`VALUE=`" when the expression contains an assignment are described in 1.7.3. Unnamed expressions like `!E(Length*Width*Height)` are described in 1.4.3.

2.2.4 Formula data structures

The formula data structure stores a Genstat formula. As explained in 1.7.3, these can be used to define the model to be fitted in a statistical analysis. Like the expression data structure (2.2.3), its main use is to give a formula as the argument of a procedure (5.3). The `FORMULA` directive which declares formulae has only the general options and parameters described in 2.1.

`FORMULA` directive

Declares one or more formula data structures.

Options

`VALUE` = *formula*	Value for all the formulae; default `*`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used by default to identify the formulae in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the formulae
`VALUE` = *formula structures*	Value for each formula
`EXTRA` = *texts*	Extra text associated with each identifier

For example:

```
FORMULA [VALUE=Drug*Logdose] Model
FORMULA BModel,Tmodel; VALUE=!F(Litter/Rat),!F(Vitamin*Protein)
```

The construction `!F(Litter/Rat)` is an example of an unnamed formula, as described in 1.4.3.

2.3 Vectors

Most Genstat directives operate on structures that store several values. The most important of these contain a list of values, which you can imagine as being arranged as a *vector* in a column. Genstat has three different types of vector: variates (2.3.1), texts (2.3.2) and factors (2.3.3). Also, the pointer structure (2.6), which stores a list of identifiers, is treated like a vector in some directives.

The directives that declare vectors all have an option called `NVALUES`, with which you can specify a scalar to define the number of values to be stored in the vector or pointer. Alternatively, you can set `NVALUES` to another text or variate; this then defines both the length of the new vectors and provides labels for use in output (3.2.1, 2:3.1.2). Finally, if you set `NVALUES` to a factor, the number of levels defines the length and its labels if available, or otherwise its levels, provide labelling.

If `NVALUES` is omitted in the declaration of a vector, Genstat takes the value or vector specified by the preceding `UNITS` statement if you have given one (2.3.4). In Genstat we call the elements of a vector its *units*. If you define values in the declaration and omit the `NVALUES` option, Genstat will deduce the appropriate setting from the number of values specified. However, it is safest to define both, since Genstat can then check that you have specified as many values as you intended. Thus, for example, if you were to type

```
    VARIATE [NVALUES=5; VALUES=1,2,3.4,5] X
```

Genstat would be able to tell you that `X` has been given only four values instead of the five that were required. Further examples are given in the subsections below.

2.3.1 Variates

The variate is probably the structure that you will use most often in Genstat. You can think of this as being just a list of numbers (a vector, in mathematical language). Variates occur for example as the response and explanatory variables in regression (Part 2 Chapter 3), as covariates and y-variables in analysis of variance (Part 2 Chapter 4), and can be used to form the matrices of correlations, similarities, or sums of squares and products required for multivariate analyses (Part 2 Chapter 6). Unnamed variates, for example `!(1,2,3,4,5)`, are described in 1.4.3. To declare a variate you use the `VARIATE` directive.

`VARIATE` directive

Declares one or more variate data structures.

Options

`NVALUES` = *scalar* or *vector*	Number of units, or vector of labels; default `*` takes the setting from the preceding `UNITS` statement, if any
`VALUES` = *numbers*	Values for all the variates; default `*`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used by default to identify the variates in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the variates
`VALUES` = *identifiers*	Values for each variate
`DECIMALS` = *scalars*	Number of decimal places for output
`EXTRA` = *texts*	Extra text associated with each identifier

MINIMUM = *scalars*	Minimum value for the contents of each structure
MAXIMUM = *scalars*	Maximum value for the contents of each structure
DREPRESENTATION = *scalars* or *texts*	Default format to use when the contents represent dates and times

For example:

```
     VARIATE Weight; EXTRA='in grams'
     VARIATE Volume,Price; VALUES=!(60,75,88),!(5,2,1.75); \
       DECIMALS=0,2
```

2.3.2 Texts

Each unit of a Genstat text structure is a string (1.4.2) which you can regard as a line of textual description. Texts can be used to label vectors and pointers (2.3 and 2.6), for captions or pieces of explanation within output (3.2.1), to store Genstat statements (1.8.2 and 5.4.3), and to store output (3.2.1). The various operations that you can perform with texts are described in 4.7. You declare texts with the TEXT directive.

TEXT directive

Declares one or more text data structures.

Options

NVALUES = *scalar* or *vector*	Number of strings, or vector of labels; default * takes the setting from the preceding UNITS statement, if any
VALUES = *strings*	Values for all the texts; default *
MODIFY = *string token*	Whether to modify (instead of redefining) existing structures (yes, no); default no
IPRINT = *string tokens*	Information to be used by default to identify the texts in output (identifier, extra); if this is not set, they will be identified in the standard way for each type of output

Parameters

IDENTIFIER = *identifiers*	Identifiers of the texts
VALUES = *texts*	Values for each text
CHARACTERS = *scalars*	Numbers of characters of the lines of each text to be printed by default
EXTRA = *texts*	Extra text associated with each identifier

For example:

```
     TEXT [NVALUES=5] Name; \
       VALUES=!T(Ferrari,Lotus,'Aston Martin',MG)
```

Unnamed texts, like that in the VALUES parameter in this example, are described in 1.4.3.

Notice that the third value has to be enclosed in single quotes as it contains a space. The rules governing when strings need to be quoted and when the quotes can be omitted are described in 1.4.2.

The text can contain any of the characters that you can generate on your computer. The text has an internal logical attribute, known as *coding*, to indicate whether it contains characters like Chinese, Korean or Thai characters, which need to be coded internally in a more complicated way than the simpler characters defined in Sections 1.3.1 to 1.3.6. (Technically they are stored as multi-byte UTF-8 characters.) Some applications may not be able to display output containing these characters successfully. You can access the coding attribute using the `GETATTRIBUTE` directive (2.11.3).

You may be unable to define all the values of a long text in its declaration, because of the restriction on the total length of a statement (1.7). One possibility then is to read the values (3.1.3). Alternatively, you could define several texts each containing a section of the full text and then use `TXCONSTRUCT` (4.7.2), `EQUATE` (4.3.1), `APPEND` (4.4.4) or `STACK` (4.4.5) to join them together. You can form a text as a progression of strings with the `TXPROGRESSION` procedure (4.7.9), or you can form the values from within the editor (4.7.10).

2.3.3 Factors

Factors are used to indicate groupings of units. The commonest occurrence is in designed experiments (Part 2 Chapter 4). For example, suppose you had 12 observations in an experiment, the first four on one treatment, the next four on a second treatment, and the last four on a third. Then you could record which treatment went with which observation by declaring a factor with the values

```
    1,1,1,1,2,2,2,2,3,3,3,3
```

Thus a factor is a vector that has only a limited set of possible values, one for each group; this limitation distinguishes factors from variates and texts. In Genstat, the groups are referred to by numbers known as *levels*. Unless otherwise specified these are the integers 1 up to the number of groups, as in our example; however, you can specify any other numbers by the `LEVELS` option of the `FACTOR` directive (see below). You can also give textual labels to the groups, using the `LABELS` option of `FACTOR`: these might, for example, be mnemonics for the biochemical names of treatments in an experiment. The full syntax of `FACTOR` is:

`FACTOR` directive

Declares one or more factor data structures.

Options

`NVALUES` = *scalar* or *vector*	Number of units, or vector of labels; default `*` takes the setting from the preceding `UNITS` statement, if any
`LEVELS` = *scalar* or *vector*	Number of levels, or series of numbers which will be used to refer to levels in the program; default `*`
`VALUES` = *numbers*	Values for all the factors, given as levels; default `*`
`LABELS` = *text*	Labels for levels, for input and output; default `*`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`REFERENCELEVEL` = *scalar*	Defines the reference level used e.g. to define the parametrization of regression models
`IPRINT` = *string tokens*	Information to be used by default to identify the factors in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the factors
`VALUES` = *identifiers*	Values for each factor, specified as levels or labels
`DECIMALS` = *scalars*	Number of decimals for printing levels
`CHARACTERS` = *scalars*	Number of characters for printing labels
`EXTRA` = *texts*	Extra text associated with each identifier

DREPRESENTATION = *scalars* or *texts*	
	Default format to use when the contents represent dates and times

Use of the VALUES parameter to assign values has the advantage that you can refer either to labels or to levels; the VALUES option lets you refer only to levels. So, to summarize, the LEVELS and LABELS options list the groups that can occur, while the VALUES option or parameter specifies which groups actually do occur, and in what pattern over the units.

Our simple explanatory example would therefore be:

```
FACTOR [LEVELS=3; VALUES=4(1...3)] Treatment
```

Other examples are:

```
FACTOR [LEVELS=!(2,4,8,16); \
  VALUES=8,4,2,16,4,2,16,8,2] Dose
FACTOR [LABELS=!T(male,female)] Sex; \
  VALUES=!T(4(male,female))
FACTOR [LEVELS=!(0,2.5,5); \
  LABELS=!T(none,standard,double)] Rate; \
  VALUES=!(0,5,2.5,5,0,2.5)
```

Notice that if we had assigned the values using the VALUES option in the second of these, we would have needed to use the (numerical) levels:

```
FACTOR [LABELS=!T(male,female); VALUES=4(1,2)] Sex
```

Conversely, in the VALUES parameter in the declaration of Rate, we can use either the labels or the levels; so the following statement gives Rate exactly the same values:

```
FACTOR [LEVELS=!(0,2.5,5); \
  LABELS=!T(none,standard,double)] Rate; \
  VALUES=!T(none,double,standard,double,none,standard)
```

When reading or printing the values of factors, you can use either the levels or labels (see the FREPRESENTATION parameter of the READ and PRINT directives: 3.1 and 3.2). The PFACLEVELS procedure allows you to print the levels, and labels (if defined), of your factors. This can be useful, for example, to check that you have defined them correctly, before either reading or printing a large data set.

Factors can be defined automatically from variates and texts by the GROUPS directive (4.6.1). You can also use factors for example to specify groups for tabulation (4.11.1), to fit parallel regression lines (2:3.3 and 2:3.7.3), and to store groupings from cluster analysis (2:6.18 and 2:6.19).

The REFERENCELEVEL option allows you to define which level of the factor is used as *reference level* if the factor is used in a regression model. By default, the first level is used, so the parameters in the model would involve comparisons of the second and subsequent levels of the factor with the first level. For example, with the factor Rate above, this would allow you to make direct assessments of the differences between standard and none, and double and none. Alternatively, you could make standard the reference level by changing the declaration to

```
FACTOR [LEVELS=!(0,2.5,5); \
  LABELS=!T(none,standard,double)] Rate; \
  VALUES=!T(none,double,standard,double,none,standard); \
  REFERENCELEVEL=2
```

2.3.4 The `UNITS` directive

`UNITS` directive

Defines an auxiliary vector of labels and/or the length of any vector whose length is not defined when a statement needing it is executed.

Option

`NVALUES` = *scalar*	Default length for vectors

Parameter

variate or *text*	Vector of labels

The `UNITS` directive can be used to define a default length which will then be used, if necessary, for any new vectors encountered later in the job. For example, in the statements

```
UNITS [NVALUES=20]
TEXT Subject
VARIATE [VALUES=0,1,2,4,8] Dlev
FACTOR [LEVELS=Dlev] Drug
VARIATE Age,Response; DECIMALS=0,2
```

the text `Subject`, the factor `Drug`, and the variates `Age` and `Response` are all defined to be of length 20. However, the length of the variate `Dlev` does not need to be set by default, but is deduced to be five from the number of values that have been specified by the `VALUES` option.

The `READ` directive (3.1) will use `UNITS` if values are to be read into a previously undeclared vector, as will the `RESTRICT` directive (4.4.1) if you use it to restrict a structure that has not yet been declared. The `UNITS` setting is also used by the `CALCULATE` directive with the `EXPAND` and `URAND` functions if their secondary argument is not specified (4.2.8 and 4.2.9).

The parameter of the `UNITS` directive allows you to specify the units structure, which is a variate or a text whose values will then be used as labels for output from regression or time-series directives, provided the vectors in the analysis have the same length as the units structure and provided also that these vectors do not have labels associated with them already.

The length of the units structure must match the value set by the `NVALUES` option if both are set. However, either one can be used to define the other. Thus, either

```
TEXT [VALUES=Sun,Mon,Tue,Wed,Thur,Fri,Sat] Day
UNITS Day
```

or

```
TEXT Day
UNITS [NVALUES=7] Day
```

would specify the default length for vectors to be seven. In the second example this default would be applied to `Day` too but, of course, its (seven) values would need to be read or defined in some other way before it could be used for labelling. If the type of the units structure has not been declared, `UNITS` will define it as a variate.

You can cancel the effect of a `UNITS` statement by

```
UNITS [NVALUES=*]
```

This means that statements that require a units structure will fail, which is the situation at the start of each job in a program. Similarly, the statement

```
UNITS *
```

cancels any reference to a units structure, but retains the default length if that has already been defined.

2.4 Matrices

A matrix stores a set of numbers as a two-dimensional array indexed by rows and columns. For example, the array

```
    1   2   3   4
    5   6   7   8
    9  10  11  12
```

is called a three-by-four matrix.

You specify the size of the matrix by saying how many rows and columns it is to have; the total number of values is obtained by multiplying the number of rows by the number of columns. In the example there are 12 values. If the numbers of rows and columns are equal the matrix is said to be square.

Any matrix can be stored as an ordinary rectangular matrix. Genstat also has special structures to store diagonal matrices (2.4.2) and symmetric matrices (2.4.3): these are needed in many statistical contexts.

You can assign values to matrices when they are declared, just as with vectors. But you must also set the size of the matrix, and it must correspond exactly to the number of values that you assign. Genstat stores the values of the matrix in row order: that is, all of the first row, followed by all of the second row, and so on. So you must assign the values in this order.

You are most likely to use matrices in multivariate analysis (Part 2 Chapter 6), where you may need them either to input data, or to save the results of an analysis. Genstat also provides many facilities for matrix calculations: for example, you can add and multiply matrices, form various patterned or standard matrices, find their inverses, and perform Choleski, eigenvalue and singular-value decompositions (4.1.3 and 4.10).

2.4.1 Rectangular matrices

The Genstat matrix structure is a rectangular array. It can be declared using the `MATRIX` directive.

`MATRIX` directive

Declares one or more matrix data structures.

Options

`ROWS` = *scalar*, *vector*, *pointer* or *text*	Number of rows, or labels for rows; default `*`
`COLUMNS` = *scalar*, *vector*, *pointer* or *text*	Number of columns, or labels for columns; default `*`
`VALUES` = *numbers*	Values for all the matrices; default `*`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used by default to identify the matrices in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the matrices
`VALUES` = *identifiers*	Values for each matrix
`DECIMALS` = *scalars*	Number of decimal places for printing
`EXTRA` = *texts*	Extra text associated with each identifier
`MINIMUM` = *scalars*	Minimum value for the contents of each structure

MAXIMUM = *scalars*	Maximum value for the contents of each structure
DREPRESENTATION = *scalars* or *texts*	Default format to use when the contents represent dates and times

You use the ROWS and COLUMNS options to specify the size of the matrix. The simplest way of doing this is to use scalars to define the numbers of rows and columns explicitly. Alternatively, you can set ROWS (or COLUMNS) to a variate, text or pointer, whose length then defines the number of rows (or columns) and whose values will then be used as labels, for example when the matrix is printed. Finally, if you specify a factor, the number of levels defines the number of rows or columns and the labels if available, or otherwise the levels, are used for labelling. Here is an example:

Example 2.4.1

```
   2  TEXT [VALUES=Beer,Lemonade,'Mineral water'] Drink
   3  VARIATE [VALUES=0.5,1.0] Quantity; DECIMALS=1
   4  MATRIX [ROWS=Drink; COLUMNS=Quantity; \
   5     VALUES=1.25,2.40,0.6,1.00,0.90,1.50] Cost
   6  PRINT Cost; DECIMALS=2

                      Cost
     Quantity          0.5          1.0
        Drink
         Beer         1.25         2.40
     Lemonade         0.60         1.00
Mineral water         0.90         1.50
```

Notice that we have set the DECIMALS parameter in the definition of the column labelling vector, Quantity, to ensure that its values are printed to one decimal place when the table Cost is printed.

In some contexts Genstat will interpret a variate as being equivalent to a matrix with a single column; this is described with each directive, such as CALCULATE (4.1.3).

2.4.2 Diagonal matrices

A square matrix that has zero entries except on its leading diagonal is called a diagonal matrix: for example,

```
2  0  0
0  1  0
0  0  3
```

Another example is the identity matrix, which has a diagonal of values equal to 1. To save space, Genstat has a special structure for diagonal matrices. You will probably use them most often to store latent roots in multivariate analysis (4.10.2, 2:6.2.1, 2:6.3.1 and 2:6.10.1). You can declare diagonal matrices using the DIAGONALMATRIX directive.

DIAGONALMATRIX directive

Declares one or more diagonal matrix data structures.

Options

ROWS = *scalar*, *vector*, *pointer* or *text*	Number of rows, or labels for rows (and columns); default *
VALUES = *numbers*	Values for all the diagonal matrices; default *
MODIFY = *string token*	Whether to modify (instead of redefining) existing

	structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used by default to identify the diagonal matrices in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the diagonal matrices
`VALUES` = *identifiers*	Values for each diagonal matrix
`DECIMALS` = *scalars*	Number of decimal places for printing
`EXTRA` = *texts*	Extra text associated with each identifier
`MINIMUM` = *scalars*	Minimum value for the contents of each structure
`MAXIMUM` = *scalars*	Maximum value for the contents of each structure
`DREPRESENTATION` = *scalars* or *texts*	Default format to use when the contents represent dates and times

Because a diagonal matrix is square, Genstat requires you to specify only the number of rows. The `ROWS` option can be set to either a scalar or a labels vector or a pointer, as in the `MATRIX` directive (2.4.1).

When you give the values of a diagonal matrix, either in a declaration or when its values are read, you should specify only the diagonal elements. (Genstat does not store the off-diagonal elements, but assumes them to be zero.) Similarly, when a diagonal matrix is printed it appears as a column of numbers; Genstat omits the off-diagonal zeros. For example:

Example 2.4.2

```
   2  DIAGONALMATRIX [ROWS=3; VALUES=2,1,3] D
   3  PRINT D

                    D

           1      2.000
           2      1.000
           3      3.000
```

2.4.3 Symmetric matrices

A symmetric square matrix is symmetric about its leading diagonal: that is, the value in column *i* of row *j* is the same as that in column *j* of row *i*. For example:

```
    1  2  3
    2  1  4
    3  4  1
```

Symmetric matrices often occur in statistics. Suppose, for example, that we have *n* random variables X_1 ... X_n. Then the covariance of X_i with X_j is the same as the covariance of X_j with X_i. The covariance matrix of the random variables is therefore symmetric: the off-diagonal elements of the matrix are the covariances (and the diagonal elements are the variances).

Because of this symmetry, Genstat stores only the diagonal elements and those below it; this is called the *lower triangle*. So you must specify only these values, whether in the declaration or in a `READ` statement (3.1). (As always, you give them in row order: so if there are *n* rows, then for the first you supply one value, for the second two, and so on.) Likewise, Genstat prints only the lower triangle in output, for example with `PRINT` (3.2).

The syntax for the declaration of symmetric matrices is as follows:

`SYMMETRICMATRIX` directive

Declares one or more symmetric matrix data structures.

Options

`ROWS` = *scalar*, *vector*, *pointer* or *text*	Number of rows, or labels for rows (and columns); default `*`
`VALUES` = *numbers*	Values for all the symmetric matrices; default `*`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used by default to identify the symmetric matrices in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the symmetric matrices
`VALUES` = *identifiers*	Values for each symmetric matrix
`DECIMALS` = *scalars*	Number of decimal places for printing
`EXTRA` = *texts*	Extra text associated with each identifier
`MINIMUM` = *scalars*	Minimum value for the contents of each structure
`MAXIMUM` = *scalars*	Maximum value for the contents of each structure
`DREPRESENTATION` = *scalars* or *texts*	Default format to use when the contents represent dates and times

The `ROWS` option defines both the number of rows and the number of columns. You can use a vector or pointer to specify row and column labels, as with `MATRIX` (2.4.1). For example:

Example 2.4.3

```
VARIATE Weight,Height,Reach
POINTER [VALUES=Weight,Height,Reach] Vars
SYMMETRICMATRIX [ROWS=Vars; VALUES=1.0,0.68,1.0,0.43,0.72,1.0] Correl
PRINT Correl

                   Correl

      Weight       1.0000
      Height       0.6800      1.0000
       Reach       0.4300      0.7200      1.0000
                   Weight      Height       Reach
```

2.5 Tables

Tables are used to store numerical summaries of data that are classified into groups. With Genstat, the classification into groups is specified by a set of factors (2.3.3). The table contains an element, called a *cell*, for each combination of the levels of the factors that classify it.

You can specify the values of a table when you declare it. More often, you may wish to calculate the values within Genstat. The `TABULATE` directive (4.11.1) allows you to summarize observations, for example from surveys. The observed values are supplied in a variate, and the levels of the factors classifying the table indicate the group to which each observed unit belongs. The table can contain, in each of its cells, either the total of the observations with the corresponding levels of the classifying factors, or perhaps the mean, or the minimum value, or the maximum value, or the variance. You can also form tables that summarize the results of surveys with multiple-response factors (4.11.8, 4.11.9 and 4.11.10). In an analysis of variance, you can save tables of means and tables of replications by the AKEEP directive (2:4.6.1). You can form tables of predictions from regression models using the PREDICT directive (2:3.3.4). Calculations with tables are described in 4.1.4. The full list of facilities available for tables is given in 4.11. Tables are declared using the TABLE directive.

TABLE directive

Declares one or more table data structures.

Options

CLASSIFICATION = *factors*	Factors classifying the tables; default *
MARGINS = *string token*	Whether to add margins (yes, no); default no
VALUES = *numbers*	Values for all the tables; default *
MODIFY = *string token*	Whether to modify (instead of redefining) existing structures (yes, no); default no
IPRINT = *string tokens*	Information to be used by default to identify the tables in output (identifier, extra, associatedidentifier); if this is not set, they will be identified in the standard way for each type of output

Parameters

IDENTIFIER = *identifiers*	Identifiers of the tables
VALUES = *identifiers*	Values for each table
DECIMALS = *scalars*	Number of decimal places for printing
EXTRA = *texts*	Extra text associated with each identifier
UNKNOWN = *identifiers*	Identifier for scalar to hold summary of unclassified data associated with each table
MINIMUM = *scalars*	Minimum value for the contents of each structure
MAXIMUM = *scalars*	Maximum value for the contents of each structure
DREPRESENTATION = *scalars* or *texts*	Default format to use when the contents represent dates and times
DATAVARIATE = *variates*	Records the identifier of the variate whose summaries are in the table
SUMMARYTYPE = *string tokens*	Records the type of summary that the table contains (counts, totals, nobservations, means, minima, maxima, variances, quantiles, sds, skewness, kurtosis, semeans, seskewness, sekurtosis); default * i.e. not recorded
PERCENTQUANTILE = *scalars*	Records the percentage points for which quantiles have been formed; default * i.e. not recorded
%MARGIN = *pointers*	Records the factors defining the margin over which the table has been converted to percentages

The example below shows a table called Classnum which stores numbers of children of each sex in the classes of a school. Here there are two factors defined in lines 2 and 3: Class with levels 1 to 5 and Sex with levels labelled 'boy' and 'girl'. The CLASSIFICATION option of the TABLE declaration (line 4) defines them to be the factors classifying the table, and the VALUES option defines a value for each of the 10 cells (two sexes × five classes) of the table. As you can see from the printed form of the table, the cells are arranged with both levels of `Sex` for `Class` 1, then both levels of `Sex` for `Class` 2, and so on. If there were three classifying factors, the table would have cells for all the levels of the third factor at level 1 of the first and second factors, then cells for all the levels of the third factor at level 1 of the first factor and level 2 of the second factor, and so on. In other words, the right-most factor in the classification rotates fastest, followed by the second from the right, and so on. This is illustrated by the second table, `Schoolnm`, which has a further factor `School` before `Class` and `Sex` in the list of classifying factors. Tables can be classified by up to nine factors.

Example 2.5a

```
FACTOR [LABELS=!T(boy,girl)] Sex
FACTOR [LEVELS=5] Class
TABLE [CLASSIFICATION=Class,Sex; \
  VALUES=15,17,29,31,34,30,33,35,28,27] Classnum
PRINT Classnum; DECIMALS=0

           Classnum
    Sex         boy        girl
  Class
      1          15          17
      2          29          31
      3          34          30
      4          33          35
      5          28          27

FACTOR [LEVELS=2] School
TABLE [CLASSIFICATION=School,Class,Sex; \
  VALUES=15,17,29,31,34,30,33,35,28,27, \
         18,16,33,31,35,36,34,33,31,32] Schoolnm
PRINT Schoolnm; DECIMALS=0

                        Schoolnm
                 Sex         boy        girl
 School        Class
      1            1          15          17
                   2          29          31
                   3          34          30
                   4          33          35
                   5          28          27
      2            1          18          16
                   2          33          31
                   3          35          36
                   4          34          33
                   5          31          32
```

A table can also have *margins*. There is then a margin for each classifying factor; this contains some sort of summary over the levels of that factor. For example, if you have a table in which the cells contain totals of the observations, you would want the marginal cells to contain totals across the levels of the factor: see the next section of the example. You can define a table to have margins when you declare it, using the `MARGINS` option of the `TABLE` directive. Or you can add margins later by the `MARGIN` directive (4.11.2), as shown in Example 2.5b.

Example 2.5b

```
MARGIN Classnum,Schoolnm
PRINT Classnum; DECIMALS=0
```

```
                 Classnum
         Sex          boy          girl       Margin
       Class
           1           15            17           32
           2           29            31           60
           3           34            30           64
           4           33            35           68
           5           28            27           55
      Margin          139           140          279
```

The margin row of `Classnum` contains the total numbers of boys and girls in the school (totalled over classes), and the margin column contains the total numbers (boys plus girls) in each class. The cell where this column and row coincide contains the total number in the school. With `Schoolnm`, there are marginal summaries over each classifying factor individually, over each pair of factors, and over all three factors. Thus the margin over a single factor is itself a two-dimensional array, classified by the other two factors, as shown in Example 2.5c.

Example 2.5c

```
  14  PRINT Schoolnm; DECIMALS=0

                                  Schoolnm
                         Sex          boy          girl       Margin
      School           Class
           1               1           15            17           32
                           2           29            31           60
                           3           34            30           64
                           4           33            35           68
                           5           28            27           55
                      Margin          139           140          279
           2               1           18            16           34
                           2           33            31           64
                           3           35            36           71
                           4           34            33           67
                           5           31            32           63
                      Margin          151           148          299
      Margin               1           33            33           66
                           2           62            62          124
                           3           69            66          135
                           4           67            68          135
                           5           59            59          118
                      Margin          290           288          578
```

Tables also have an associated scalar which collects a summary of all the observations for which any of the classifying factors has a missing value; these observations cannot be assigned to any cell of the table itself. This scalar is known as the *unknown cell* of the table. It can be given an identifier, so that you can refer to it, using the `UNKNOWN` parameter of the `TABLE` directive.

The `IPRINT` option of `TABLE` has a special setting, `associatedidentifier`, that refers to the "associated identifier" of the table, if available. This is the identifier of the data variate from which the summaries in the table have been formed.

The attribute of the table that records its data variate is set automatically when a table of summaries is formed by the `TABULATE` directive (4.11.1). If you have formed the summaries in some other way, you can use the `DATAVARIATE` parameter to record the relevant variate yourself. The `SUMMARYTYPE` parameter can set an attribute recording the type of summary that the table contains, and the `PERCENTQUANTILE` parameter can set an attribute recording the corresponding percentage if the table contains quantiles. (These are alse set automatically for tables formed by `TABULATE`.) The `%MARGIN` parameter can be set to a pointer of factors defining the margin of the table over which it has been converted to percentages; the `PERCENT` procedure (4.1.3) will set this attribute automatically. If any of these parameters is not set, the default is to leave the corresponding attribute of the table unchanged. To clear the existing value of one of these attributes, you can put a missing value into the corresponding parameter setting. For example

```
    TABLE Tab; DATAVARIATE=*; SUMMARYTYPE=*; PERCENTQUANTILE=*;\
        %MARGIN=*
```

clears all these attributes for the table `Tab`.

2.6 Pointers

A pointer is a data structure that points to other structures: that is, each of its elements is the identifier of some other Genstat data structure. You use pointers in Genstat wherever you have to specify a collection of structures; for example in `EQUATE` (4.3.1), `COMBINE` (4.11.4), in some functions (4.2.3), and for a data matrix specified via the variates forming its columns (2:6.2). You can use them as a convenient means of specifying a list of structures (1.5.4), and they are also involved in the use of suffixed identifiers (see below). You can declare pointers using the `POINTER` directive.

`POINTER` directive

Declares one or more pointer data structures.

Options

`NVALUES` = *scalar* or *text*	Number of values, or labels for values; default `*`
`VALUES` = *identifiers*	Values for all the pointers; default `*`
`SUFFIXES` = *variate* or *scalar*	Defines an integer number for each of the suffixes; default `*` indicates that the numbers 1,2,... are to be used
`CASE` = *string token*	Whether to distinguish upper and lower case in the labels of the pointers (`significant`, `ignored`); default `sign`
`ABBREVIATE` = *string token*	Whether or not to allow the labels to be abbreviated (`yes`, `no`); default `no`
`FIXNVALUES` = *string token*	Whether or not to prohibit automatic extension of the pointers (`yes`, `no`); default `no`
`RENAME` = *string token*	Whether to reset the default names of elements of the pointer if they do not have their own identifiers (`yes`, `no`); default `no`
`MODIFY` = *string token*	Whether to modify (instead of redefining) existing structures (`yes`, `no`); default `no`
`IPRINT` = *string tokens*	Information to be used by default to identify the pointers in output (`identifier`, `extra`); if this is not set, they will be identified in the standard way for each type of output
`EXTEND` = *string token*	Whether to extend (instead of redefining) an existing pointer (`yes`, `no`); default `no`

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the pointers
`VALUES` = *pointers*	Values for each pointer
`EXTRA` = *texts*	Extra text associated with each identifier

Thus, for example,

```
    POINTER [VALUES=Yield,Costs,Profit] Info
```

sets up a pointer `Info` with values `Yield`, `Costs` and `Profit`. These three are themselves data structures, which can be assigned values, operated on, and so forth. You can refer to individual elements of pointers by suffixes, enclosed in square brackets (1.4.3): so `Info[3]` is `Profit`, and `Info[1,2]` is the list of structures `Yield`, `Costs`. Thus if `Yield` held the values `5.6` and `6.1`, then

```
PRINT Info[1]
```

would print the values of `Yield`, as shown below:

Example 2.6

```
   2  VARIATE [NVALUES=2] Yield,Costs,Profit
   3  READ Yield,Costs,Profit

  Identifier   Minimum      Mean   Maximum    Values   Missing
       Yield     5.600     5.850     6.100         2         0
       Costs      1200      1365      1530         2         0
      Profit     455.0     537.5     620.0         2         0

   5  POINTER [VALUES=Yield,Costs,Profit] Info
   6  PRINT Info[1]

       Yield
       5.600
       6.100
```

In fact, when Genstat meets a suffixed identifier, it usually sets up a pointer automatically if necessary. For example if your program contains a suffixed identifier `Data[4]`, Genstat first checks whether or not a pointer called `Data` already exists and, if not, creates it; then if there is no element for suffix `4` it creates one. If the pointer `Data` already exists but does not have a fourth element, then an appearance of `Data[4]` automatically extends `Data`. So you can add elements to pointers without redeclaring them.

If, however, you do not want a pointer to be extended automatically, you should declare it explicitly and set option `FIXNVALUES=yes`. For example, if you had specified

```
POINTER [NVALUES=4; FIXNVALUES=yes] Data
```

the statement

```
CALCULATE Data[5] = Data[1] + Data[2]
```

would be faulted. This can be useful if you know in advance all the suffixes that may occur, and want to guard against mistyped suffixes.

The suffixes need not run from 1, nor be a complete list, although they must be integers; if you give a decimal number it will be rounded to the nearest integer (for example, -27.2 becomes -27). You specify the list of suffixes that you require by the `SUFFIXES` option; if you omit this, they are assumed to run from 1 up to the number of values.

You can also label the elements of pointers by supplying a text in the `NVALUES` option (2.3): for example

```
POINTER [NVALUES=!T(workstations,PCs,laptops)] Sales
```

allows you to refer to `Sales['PCs']`, `Sales['laptops','workstations']`, and even to `Sales[1,2,'laptops']`. The suffix list within the square brackets is a list of identifiers, so the strings must be quoted: they are then treated as unnamed texts each with a single value (1.4.3). By default, the case (small letters or capitals) of the textual suffixes is significant, so `Sales['Laptops']` would not be the same as `Sales['laptops]`. However, you can set option `CASE=ignored` to indicate that Genstat should ignore the case of the letters in a textual suffix. You can also set the `ABBREVIATE` option to `yes` to allow each suffix to be abbreviated to the minimum number of letters required to distinguish it from the earlier suffixes in the list. So, if instead you define

```
POINTER [NVALUES=!T(workstations,PCs,laptops); \
  CASE=ignored; ABBREVIATE=yes] Sales
```

you can refer to `Sales['PCs']` as, for example, `Sales['pcs']`, or `Sales['pc']`, or `Sales['P']`, and so on. By default `CASE=significant` and `ABBREVIATE=no`.

The identifiers in a suffix list can be of scalars, variates or texts; this of course includes numbers and strings as unnamed scalars and texts respectively. If one of these structures contains several values, it defines a sub-pointer: for example `Info[!(3,2)]` is a pointer with two elements, `Profit` and `Costs`. You can also give a null list to mean all the elements of the pointer: for example `Info[]` is `Yield,Costs,Profit`. You must be careful not to confuse a sub-pointer with a list of some of the elements of a pointer: for example `Info[!(3,2)]` is a single pointer with two elements, whereas `Info[3,2]` is a list of the two structures `Profit` and `Costs`.

As mentioned above, a pointer can be extended automatically to include a new suffix, if that suffix is used with the pointer in your program. However, it is not possible to extend the pointer automatically to include a new label, as Genstat would not know which suffix to give it and an automatic choice could lead to errors or confusion. So, the `POINTER` directive has an option `EXTEND` which can be set to `yes` to do this explicitly. The pointer elements that are defined are then added to the existing elements of the pointer. So, we could add additional labels to the pointer `Employee`, above, by the statements

```
TEXT [VALUES=netbooks,printers] Newlabs
VARIATE [VALUES=4,5] Newsuffs
POINTER [NVALUES=Newlabs; SUFFIXES=Newsuffs; EXTEND=yes] Sales
```

adds `Sales['netbooks']` as suffix 4, and `Sales['printers']` as suffix 5. If you do not specify a label, the new suffix is still added (but unlabelled). If you do not specify a suffix, the new label is given a suffix of one plus the largest suffix already in the pointer. When `EXTEND=yes`, the `EXTRA` parameter and the `CASE`, `ABBREVIATE`, `FIXNVALUES`, `MODIFY` and `IPRINT` options are ignored.

Elements of pointers can themselves be pointers, allowing you to construct trees of structures. For example

```
VARIATE A,B,C,D,E
POINTER R; VALUES=!P(D,E)
& S; VALUES=!P(B,C)
& Q; VALUES=!P(A,S)
& P; VALUES=!P(Q,R)
```

defines the tree

```
               P
         /         \
     Q                R
   /   \            /   \
A        S        D       E
       /  \
     B      C
```

You can refer to elements within the tree by giving several levels of suffixes: for example `P[2][1]` is `R[1]` which is `D`; `P[2,1][1,2]` is `(R,Q)[1,2]` or `D,E,A,S`. The special symbol # (1.3.6 and 1.5.4) allows you to list all the structures at the ends of the branches of the tree: `#P` replaces `P` by the identifiers of the structures to which it points (`Q` and `R`); then, if any of these is a pointer, it replaces it by its own values, and so on. Thus `#P` is the list `A,B,C,D,E`.

The `RENAME` option allows you to control what identifier is used in output when a structure in the pointer already belongs to another pointer, but does not have an identifier of its own. For example, in

```
POINTER [NVALUES=2] X
POINTER [VALUES=A,X[1],B] Y
```

```
VARIATE [VALUES=1,2,3,4] Y[]
PRINT Y[]
```

the second element of `Y` has no identifier of its own but was originally defined as the first element of the pointer `X`. Genstat labels the output from a structure like this using the identifier of the pointer with which it was first associated. So, the output will show the identifiers `A`, `X[1]` and `B`. However, you can set option `RENAME=yes` when `Y` is defined

```
POINTER [VALUES=A,X[1],B; RENAME=yes] Y
```

to request that the pointer `Y` takes precedence over earlier definitions. The identifiers would then become `A`, `Y[2]` and `B`.

2.7 Compound structures

You can use the pointer structure (2.6) to group together related data structures, so that you can refer to them as a single structure. Some Genstat directives expect standard combinations of data structures for their input or output; in these cases you use special pointers called *compound structures*. These differ from ordinary pointers in that they have a fixed number of elements which must be of the correct types, and must form a consistent set (in terms of their sizes and so on).

You can refer to elements of these structures in exactly the same way as the elements of pointers: for example if `L` is an LRV (2.7.1) then `L` refers to a set of structures `L[1], L[2], L[3]`. The suffixes run from 1 upwards, and Genstat does not allow you to change that. Neither can you change the labels that Genstat gives to the structures; details of these labels come later in this section. However, like a pointer defined with `CASE=ignored`, the labels are not case sensitive; so Genstat will recognize the label in either upper case or lower case, or any mixture.

You can give the individual elements of a compound structure identifiers in their own right, just as with pointers. Indeed, you can use all the features of pointer syntax: for example, you can use the null list, or the substitution symbol #, to list all the elements of the structure (2.6).

When you declare a compound structure, you conveniently declare, simultaneously and automatically, a whole collection of structures. At the same time you ensure that they match the requirements of whatever form of analysis you want to use.

2.7.1 The LRV structure

The LRV structure is used to store latent roots and vectors resulting from the decomposition of a matrix (4.10.2), or produced in multivariate analysis (Part 2 Chapter 6). You need not store all the latent roots; usually Genstat will select the largest ones. The LRV structure points to three structures (identified by their suffixes):

`[1]` or `['Vectors']` is a matrix whose columns are the latent vectors (the word `"Vector"` is used here in its mathematical sense rather than in the more specific Genstat sense; in fact, latent vectors are most conveniently stored in matrices rather than in Genstat vectors);

`[2]` or `['Roots']` is a diagonal matrix whose elements are the latent roots;

`[3]` or `['Trace']` is a scalar holding the trace of the matrix, which is the sum of all its latent roots.

As mentioned earlier, the labels can be specified in either lower or upper case, or any mixture. To declare an LRV you use the `LRV` directive.

LRV directive

Declares one or more `LRV` data structures.

Options

`ROWS` = *scalar*, *vector* or *pointer*	Number of rows, or row labels, for the matrix; default *
`COLUMNS` = *scalar*, *vector* or *pointer*	

Number of columns, or column labels, for matrix and diagonal matrix; default *

Parameters

IDENTIFIER = *identifiers*	Identifiers of the LRVs
VECTORS = *matrices*	Matrix to contain the latent vectors for each LRV
ROOTS = *diagonal matrices*	Diagonal matrix to contain the latent roots for each LRV
TRACE = *scalars*	Trace of the matrix

The length of each latent vector is specified by the ROWS option; this then defines the number of rows in the 'VECTORS' matrix. The COLUMNS option defines the number of latent roots to be stored; this is also the number of latent vectors, and so indicates the number of columns in the 'VECTORS' matrix and the number of elements in the 'ROOTS' matrix. If you do not specify the number of columns Genstat will set it to be the same as the number of rows. The value of COLUMNS can be less than the value of ROWS; however, it must not exceed than that of ROWS, otherwise Genstat gives an error diagnostic. Row and column labels can be defined, as in the declaration of matrices (2.4).

You can specify identifiers for the three individual elements of the LRV by using the VECTORS, ROOTS and TRACE parameters. If you have declared them already they must be of the correct type (and you can also have given them values). If you have given these identifiers row or column settings, then these will be used for the LRV declaration and must match any of the corresponding options of LRV that you choose to set.

Example 2.7.1 declares an LRV, and then forms its values (see 4.10.2).

Example 2.7.1

```
   2  POINTER [VALUES=stem,leaf,root,petal,pollen] Vars
   3  SYMMETRICMATRIX [ROWS=Vars] Symm
   4  READ Symm

    Identifier   Minimum      Mean   Maximum     Values   Missing
          Symm   -0.9820    0.1974     1.000         15         0

  10  PRINT Symm

                   Symm

         stem     1.0000
         leaf    -0.6550     1.0000
         root    -0.9450     0.8660     1.0000
        petal    -0.7560     0.0000     0.5000     1.0000
       pollen     0.5000    -0.9820    -0.7560     0.1890     1.0000
                    stem       leaf       root      petal     pollen

  11  LRV [ROWS=Vars;COLUMNS=2] Latent; VECTORS=Lvecs
  12  FLRV Symm; Latent
  13  PRINT Latent['Vectors','Roots']

                  Lvecs
                      1          2
         Vars
         stem     0.4875    -0.3372
         leaf    -0.4875    -0.3372
         root    -0.5335     0.0770
        petal    -0.2227     0.7383
       pollen     0.4366     0.4707


                          1          2
Latent['Roots']       3.482      1.518
```

2.7.2 The SSPM structure

The SSPM structure stores a matrix of corrected sums of squares and products, and associated information, as used for regression (Part 2 Chapter 3) and some multivariate analyses (Part 2 Chapter 6). You can form values for SSPM structures by the `FSSPM` directive (4.10.3). However, most multivariate and regression analyses can be done without declaring and forming an SSPM explicitly.

An SSPM comprises four structures (identified by their suffixes).

`[1]` or `['Sums']` is a symmetric matrix containing the sums of squares and products. The number of rows and columns of this matrix will equal the number of parameters defined by the expanded terms list: that is, the number of variates plus the number of dummy variates generated by the model formula. (See the `TERMS` directive: 3:3.2.2.)

`[2]` or `['Means']` is a variate containing the mean for each variate or dummy variate.

`[3]` or `['Nunits']` is a scalar holding the total number of units used in constructing the sums of squares and products matrix. If the SSPM is weighted, this scalar will hold the sum of the weights.

A within-group SSPM has one additional element:

`[4]` or `['Wmeans']` is a pointer, pointing to variates holding within-group means. There is one variate for each row of the `'Sums'` matrix plus one extra. They are all of the same length, namely the number of levels of the `GROUPS` factor. The extra variate holds counts of the number of units in each group.

As mentioned earlier, the labels can be specified in either lower or upper case, or any mixture. The syntax for the declaration of SSPM structures is as follows:

`SSPM` directive

Declares one or more SSPM data structures.

Options

`TERMS` = *formula*	Terms for which sums of squares and products are to be calculated; default `*`
`FACTORIAL` = *scalar*	Maximum number of vectors in a term; default 3
`FULL` = *string token*	Full factor parameterization (`yes`, `no`); default `no`
`GROUPS` = *factor*	Groups for within-group SSPMs; default `*`
`DF` = *scalar*	Number of degrees of freedom for sums of squares; default `*`

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the SSPMs
`SSP` = *symmetric matrices*	Symmetric matrix to contain the sums of squares and products for each SSPM
`MEANS` = *variates*	Variate to contain the means for each SSPM
`NUNITS` = *scalars*	Number of units or sum of weights for each SSPM
`WMEANS` = *pointers*	Pointers to variates of group means for each SSPM

The `TERMS` option defines the model for whose components the sums of squares and products are to be calculated. In the simplest case the model is just a list of variates, but you can use more complex model formulae, involving variates and factors; this is done in conjunction with the `FACTORIAL` and `FULL` options. Details of how formulae are interpreted in regression are given in 2:3.3.1.

You can form a within-group matrix of sums of squares and products by specifying the relevant factor with the `GROUPS` option.

Sometimes you may already have calculated values for the matrix of sums of squares and products. You can then assign them to the component structures of the SSPM for example by `READ` (3.1). You would still, however, need to set the number of degrees of freedom associated with the matrix, and for that you use the `DF` option.

The parameter lists let you specify identifiers for the four components of an SSPM. You can have declared them previously (and you can have given them values), but if so they must be of the correct type.

Example 2.7.2 shows the declaration and formation (4.10.3) of an SSPM.

Example 2.7.2

```
   2  READ [SETNVALUES=yes] V[1...5]

    Identifier   Minimum      Mean   Maximum    Values   Missing
          V[1]     1.000     2.667     4.000         3         0
          V[2]    0.0000     2.000     4.000         3         0
          V[3]     1.000     3.000     7.000         3         0
          V[4]    0.0000    0.6667     1.000         3         0
          V[5]    0.0000     1.333     3.000         3         0

   6  SSPM [TERMS=V[1...5]] Ssp
   7  FSSPM [PRINT=sspm] Ssp


Degrees of freedom
------------------

Sums of squares:  2
Sums of products: 1


Sums of squares and products
----------------------------

    V[1]     1      4.6667
    V[2]     2     -4.0000      8.0000
    V[3]     3    -10.0000     12.0000     24.0000
    V[4]     4     -1.3333      0.0000      2.0000      0.6667
    V[5]     5      2.3333     -6.0000     -8.0000      0.3333      4.6667
                         1           2           3           4           5


Means
-----

    V[1]     1       2.667
    V[2]     2       2.000
    V[3]     3       3.000
    V[4]     4      0.6667
    V[5]     5       1.333


Number of units used
--------------------

            3
```

2.7.3 The TSM structure

The TSM structure stores a time-series model which you can use in Box-Jenkins modelling of time series (see Part 2 Chapter 7). The information that you give to specify the model is stored in two variates, called the *orders* and the *parameters*; an optional third variate contains *lags*. A complete description of how these structures are defined and assigned values is given in Part 2 Chapter 7.

The elements of a TSM are:

`[1]` or `['Orders']`;

[2] or ['Parameters'];
[3] or ['Lags'].
As mentioned earlier, the labels can be specified in either lower or upper case, or any mixture.
To declare a TSM you use the TSM directive.

TSM directive
Declares one or more TSM data structures.

Option

MODELTYPE = *string token*	Type of model (arima, transfer); default arim

Parameters

IDENTIFIER = *identifiers*	Identifiers of the TSMs
ORDERS = *variates*	Orders of the autoregressive, integrated and moving-average parts of each TSM
PARAMETERS = *variates*	Parameters of each TSM
LAGS = *variates*	Lags, if not default

The TSM directive sets up a compound structure pointing to the variates that will later be used to define the model. You set the type of model by the MODELTYPE option. You can use the parameters of TSM to supply previously declared identifiers as the elements of the TSM, just as with the LRV and SSPM. In this way you can specify a variate of lags, to give the TSM three elements rather than the default of two.
Here are some examples:

```
TSM [MODELTYPE=arima] T1
TSM [MODELTYPE=transfer] T2; ORDERS=!(1,0,1)
TSM T3; ORDERS=O; PARAMETERS=P; LAGS=L
```

2.7.4 Customized compound structures
The STRUCTURE directive allows you to define your own types of compound data structure, and specify constraints on the structures that they may contain.

STRUCTURE directive
Defines a compound data structure.

Options

NAME = *text*	Single-valued text defining a name for the type of structure, which must not clash with the name of any existing type of structure
STRUCTURELIST = *string token*	Whether or not the structure consists of a list (of any length) of structures of the same type or types (yes, no); default no

Parameters

LABEL = *texts*	Single-valued texts defining the labels of the elements of the structure
SUFFIX = *scalars*	Suffix numbers for the elements; default assumes the numbers 1, 2 ...
TYPE = *texts*	Texts defining the allowed types for each element

COMPATIBLE = *texts*	Defines aspects to check for compatibility with the first element

The STRUCTURE directive allows you to define customized compound data structures for use, for example, in procedures. The NAME option supplies a single-valued text of up to 16 characters to define the name to be used for the new "type" of data structure. This can then be used as a setting for the TYPE parameter in either the OPTION or PARAMETER directives within a procedure, to indicate that the option or parameter concerned must be supplied with this type of structure. The case of the letters in the name is not significant, so they can be in capitals, or in lower case, or in any mixture.

The parameters of the directive define the contents of the structure. The LABEL parameter lists the labels to be used with each element of the structure, and the SUFFIX parameter lists the corresponding suffix numbers (by default these are the numbers 1, 2, etc.). The TYPE parameter allows you to define the types of structure that are allowed in each element (which may be any of the standard Genstat data structures, or other customized types), and the COMPATIBLE parameter allows you to define aspects that must be compatible with the first element of the structure similarly to the COMPATIBLE parameter of the OPTION and PARAMETER directives (5.3.2). These are checked when the structure is declared, and when it is used as an option or parameter setting of a procedure that requests that type.

For example, we could define a 'complex number' type by

```
STRUCTURE [NAME='complex number'] 'real','imaginary'; \
  TYPE='scalar'
```

Structures of the new type can be defined using the DECLARE directive.

DECLARE directive

Declares one or more customized data structures.

Options

TYPE = *text*	Single-valued text defining the type of structure to declare
MODIFY = *string token*	Whether to modify (instead of redefining) existing structures (yes, no); default no

Parameters

IDENTIFIER = *identifiers*	Identifiers of the structures
VALUES = *pointers*	Values for each structure
EXTRA = *texts*	Extra text associated with each identifier

DECLARE is used to set up compound data structures of a customized type, defined earlier using the STRUCTURE directive. So we can declare a complex number C (as defined above) by

```
DECLARE [TYPE='complex number'] C; VALUES=!p(3,2)
```

The VALUES parameter allows values to be defined for the structure, similarly to the VALUES parameter of the POINTER directive. So, here, the real part of the number C['real'] is given the value 3, and the imaginary part C['imaginary'] has the value 2. The EXTRA parameter is also used as in the POINTER directive, allowing extra text to be associated with the structure for annotation, and the MODIFY option allows an existing structure to be modified.

The elements of the compound structure can be referred to like those of an ordinary pointer declared using the POINTER directive (2.6) with options CASE=ignored, ABBREVIATE=yes and FIXNVALUES=yes. So, the labels can be given in either upper or lower case or in any mixture, and each can be abbreviated to the minimum number of characters required to distinguish it from the previous labels. So the imaginary part of the complex number above could, for example, be referred to as `C['imaginary']` or `C['IMAGINARY']` or simply `C['i']`.

2.8 Tree structures

A tree structure is like a real tree, which starts from a root and then splits into branches, except that it is usually viewed as growing downwards instead of upwards. The branch-points in the tree are known as *nodes*, with the initial node being called the *root* (as in a real tree). There is also a node at the end of each branch, known as its *terminal node*.

In Genstat a tree is similar to a pointer, with an element for each node. These elements are the identifiers of data structures which can be used to store information about the nodes. Usually the data structures will be pointers, so that several pieces of information can be stored for each node, but the precise contents depend on the type of tree.

Each node thus has a number, corresponding to the index of its element in the tree. The root is always numbered one, and this is the only node that the tree contains when it is declared by `the TREE` directive. Further nodes can be added by directives `BGROW` (4.12.3) or `BJOIN` (4.12.5), which form branches from a terminal node or join another tree to a terminal node, respectively. The converse process of cutting a tree at a defined node and discarding the nodes and information below it is provided by the `BCUT` directive (4.12.4). The numbers of the subsequent nodes can be obtained from the functions that are provided to navigate around a tree (4.2.11). There are also several utility procedures for trees (4.12) as well as special-purpose procedures for classification trees (2:6.20), identification keys (2:6.21) and regression trees (2:3.9).

`TREE` directive

Declares one or more tree data structures and initializes each one to have a single node known as its root.

No options

Parameter

`IDENTIFIER` = *identifiers*	Identifiers of the trees

2.9 Save structures

Genstat has several special-purpose structures for saving the information from an analysis. These cannot be declared explicitly, but are defined automatically by the directives that perform the analysis. For example, the ASAVE structure (2:4.6) can be defined by `ANOVA` and then used by `ADISPLAY` or `AKEEP`.

```
ANOVA Gain; SAVE=Gsave
ADISPLAY [PRINT=residuals] SAVE=Gsave
AKEEP [SAVE=Gsave] Source.Amount; MEANS=Meangain
```

In many cases the structure need not be mentioned explicitly. For example, Genstat automatically stores the ASAVE structure from the last y-variate analysed by `ANOVA`, and `ADISPLAY` and `AKEEP` will use this by default if no other ASAVE structure is specified. Save structures are also available from regression and generalized linear models (RSAVE, 2:3.1.1), `REML` (VSAVE, 2:5.3.1), time series (TSAVE, 2:7.3.3), and to store the environment for high-resolution graphics (DSAVE) They can all be accessed using the `GET` (5.6.2), and reset using the `SET` directive (5.6.1).

2.10 Deleting, renaming and duplicating data structures

The section describes commands to manage your data structures: deleting those that you no longer need (2.10.1), giving them new names (2.10.2), or creating new structures with the same attributes – and perhaps also values – as those of existing structures (2.10.3 and 210.4).

2.10.1 The DELETE directive

DELETE directive

Deletes the attributes and values of structures.

Options

REDEFINE = *string token*	Whether or not to delete the attributes of the structures so that the type etc can be redefined (yes, no); default no
LIST = *string token*	How to interpret the list of structures (inclusive, exclusive, all); default incl
PROCEDURE = *string token*	Whether the list of identifiers is of procedures instead of data structures (yes, no); default no
NSUBSTITUTE = *scalar*	Number of times *n* to substitute a dummy in order to determine which structure to delete; default * i.e. full substitution
REMOVE = *string token*	Whether or not to remove the structures from Genstat completely i.e. to delete their identifiers as well as their attributes and values (yes, no); default no

Parameter

identifiers	Structures whose values (and attributes, if requested) are to be deleted

Genstat stores the values and attributes of data structures in internal arrays. These arrays expand automatically according to the amount of data – until they reach the limitations of your computer. However, once you have finished with a structure, it may still be sensible to delete its values. Genstat should then execute more efficiently as it will need to keep track of less information. You can also delete the attributes of data structures. This can be worthwhile merely to save further space, but the main advantage is that the structures can then be redefined to be of different types. Both of these actions can be carried out using the DELETE directive.

Each time that DELETE is used, Genstat will also remove any unnamed structures that are no longer required, and recover any space that has been used for temporary storage. This sort of tidying of workspace will happen automatically if Genstat sees in time that space is becoming short. However, to avoid unnecessary computation, this does not occur after every statement. Thus, if the space appears to be exhausted, it may be worth using DELETE, even if you have no named structures to delete.

The REDEFINE option controls whether the attributes of the structures are deleted as well as their values. If REDEFINE is set to yes, the only information that is still stored is the identifier and the internal reference number of the structure. Alternatively, you can set option REMOVE=yes to delete the identifier and reference number as well as the attributes and values, so that no trace of the structure remains.

With the defaults, REDEFINE=no and REMOVE=no, only the values of the structures are deleted. For example, suppose we have defined a variate Dose by

```
    VARIATE [VALUES=0,0,2,2,4,4] IDENTIFIER=Dose
```

This gives Dose the values 0, 0, 2, 2, 4 and 4. If we then put

```
DELETE Dose
```

only the values of Dose are deleted; so we could now assign a new set: for example

```
READ Dose
2 4 0 4 2 0 :
```

Dose remains a variate but now has the values 2, 4, 0, 4, 2 and 0.

Alternatively, if we set REDEFINE=yes in the above example, we could then redefine Dose as (for example) a text with seven values.

```
DELETE [REDEFINE=yes] Dose
TEXT [VALUES=none,double,standard,double,none,\
  standard,none] Dose
```

Once you have defined the type of a structure in a job (as variate, factor or whatever), you cannot redeclare it as a structure of any other type unless you have first used DELETE to delete its values and attributes. The only exceptions to this rule, with their own REDEFINE options, are the DUPLICATE directive and the GROUPS directive (which allows a variate or text to be redefined as a factor).

The NSUBSTITUTE option is relevant when the list of structures to delete contains dummies. The default setting, missing value, requests all dummies to be replaced by the structures to which they point (so that those are the structures that are deleted). NSUBSTITUTE allows you to delete dummies instead. If you set NSUBSTITUTE=0, no dummies are substituted. So the deleted structures are the actual dummies that you have listed. A positive setting n>0 is useful if you have dummies pointing to other dummies, in a chain. Each dummy in the list is then substituted n times in order to determine which structure in each chain to delete. For example, suppose we have

```
DUMMY A; VALUE=B
& B; VALUE=C
SCALAR c; VALUE=1
```

Then

```
DELETE A
```

would delete the scalar C (as this is the structure to which the dummies A and B finally point), but

```
DELETE [NSUBSTITUTE=1] A
```

would delete B, and

```
DELETE [NSUBSTITUTE=0] A
```

would delete A itself.

The LIST option defines how the parameter list is to be interpreted. With the default setting, LIST=inclusive, attributes or values are deleted only for the structures in the list (as well those of any unnecessary unnamed structures). If there is no parameter list, then only unnamed structures are deleted. LIST=exclusive means that the parameter list is the complement of the set of structures that are deleted: that is, all named or unnamed structures that are not in the list are deleted. LIST=all causes the attributes or values of all structures to be deleted. Thus, if LIST=all, any parameter list is ignored; and LIST=exclusive with no parameter is equivalent to LIST=all.

2.10.2 The RENAME directive

RENAME directive

Assigns new identifiers to data structures.

No options

Parameters

OLDIDENTIFIER = *identifiers*	Specifies the data structures to rename
NEWIDENTIFIER =*identifiers*	Specifies a new identifier for each data structure

---

RENAME allows you to assign a different identifier to a data structure. For example, if you put

```
    RENAME OLDIDENTIFIER=A; NEWIDENTIFIER=B
```

the data structure previously known as A would be renamed to have the identifier B, and the data structure previously known as B would lose its identifier and become unnamed. The identifier A would then no longer belong to anyone (and could if required be reused).

In the simplest situations, like Example 2.10.2 below, the first appearance of the new identifier will be in the RENAME command. So there will be no consequences from the fact that the "orphan" data structure that it previously identified becomes unnamed.Here the factors N and S are renamed to have the new identifiers Nitrogen and Sulphur, which had not been used before. One beneficial side effect to notice is that the renaming carries over into al lthe data structures that use N and S. So the table Meanyield is now classified by Nitrogen and Sulphur (but otherwise unchanged).

Example 2.10.2

```
" define factors N and S, and an N x S table "
VARIATE [VALUES=0,180,230] Nlev
&       [VALUES=0,10,20,40] Slev
FACTOR  [LEVELS=Nlev; VALUES=4(0,180,230)] N; DECIMALS=0
&       [LEVELS=Slev; VALUES=(0,10,20,40)3] S; DECIMALS=0
TABLE   [CLASSIFICATION=N,S; VALUES=0.560,0.770,0.524,0.552,\
        0.894,1.289,1.525,1.545, 1.032,1.404,1.454,1.700] MeanYield
PRINT   MeanYield

           MeanYield
      S            0          10          20          40
      N
      0        0.560       0.770       0.524       0.552
    180        0.894       1.289       1.525       1.545
    230        1.032       1.404       1.454       1.700


" print to show the original values of N and S "
PRINT   N,S

      N           S
      0           0
      0          10
      0          20
      0          40
    180           0
    180          10
    180          20
    180          40
    230           0
    230          10
    230          20
    230          40
```

```
 12  " rename N to Nitrogen, and S to Sulphur "
 13  RENAME  N,S; NEWIDENTIFIER=Nitrogen,Sulphur
 14  " print to show that N has become Nitrogen, and S has become Sulphur "
 15  PRINT   Nitrogen,Sulphur

   Nitrogen     Sulphur
          0           0
          0          10
          0          20
          0          40
        180           0
        180          10
        180          20
        180          40
        230           0
        230          10
        230          20
        230          40

 16  " notice the renaming carries over to the classification of MeanYield "
 17  PRINT   MeanYield

              MeanYield
     Sulphur          0          10          20          40
    Nitrogen
           0      0.560       0.770       0.524       0.552
         180      0.894       1.289       1.525       1.545
         230      1.032       1.404       1.454       1.700
```

If the identifier has already been used, the orphan data structure will be deleted, unless it is found to belong to another (named) data structure. So, for example, in the program

```
SCALAR B; VALUE=1
POINTER [VALUES=B] Q
RENAME OLDIDENTIFIER=A; NEWIDENTIFIER=B
```

the scalar 1 would survive as the first element of the pointer `Q`. So it could still be referred to as `Q[1]`, although of course no longer as `B`. You would get the same effect be specifying

```
RENAME OLDIDENTIFIER=A; NEWIDENTIFIER=Q[1]
```

as `RENAME` looks only for the (named) identifier of the data structure specified by `NEWIDENTIFIER`. So, in this case, `A` takes over the identifier `B` of `Q[1]`. If `Q[1]` did not have a separate identifier of its own, `A` would become unnamed. (So this provides a way of removing the identifier of a pointer element.)

You can also specify a pointer element for the setting of `OLDIDENTIFIER` and, again, `RENAME` will operate only on its identifier (if it has one). For example, in the program

```
SCALAR C; VALUE=7
POINTER [NVALUES=2] P
RENAME OLDIDENTIFIER=P[1]; NEWIDENTIFIER=C
```

the pointer element `P[1]` gains the identifier `C`, and so can be referred to as `C` in future (as well as `P[1]`).

So, to summarize, for the data structures specified by both the `OLDIDENTIFIER` and `NEWIDENTIFIER` parameters, `RENAME` ignores any memberships that they may have of pointers, or e.g. as classifying factors of a table, or as levels or labels vectors of factors. It operates only on their own identifiers, reassigning the one (if any) belonging to the `NEWIDENTIFIER` data structure to become the identifier of the data structure specified by the `OLDIDENTIFIER` parameter.

Note that, if either `OLDIDENTIFIER` or `NEWIDENTIFIER` is set to a dummy, `RENAME` will operate on the data structure to which it points, not on the dummy itself (i.e. dummies are always substituted). So, this allows you to rename data structures in your main program from inside a procedure.

A final point is that, if your new name is stored inside a text, you may find the SETNAME procedure more convenient than RENAME; details are in the *Genstat Reference Manual, Part 3 Procedures*.

2.10.3 The DUPLICATE directive

DUPLICATE directive

Forms new data structures with attributes taken from an existing structure.

Options

ATTRIBUTES = *string tokens*	Which attributes to duplicate (all, nvalues, values, nlevels, levels, labels (of factors or pointers), extra, decimals, characters, rows, columns, classification, margins, suffixes, minimum, maximum, restriction, referencelevel); default all
REDEFINE = *string token*	Whether or not to delete the attributes of the new structures beforehand so that their types can be redefined (yes, no); default no

Parameters

OLDSTRUCTURE = *identifiers*	Data structures to provide attributes for the new structures
NEWSTRUCTURE = *identifiers*	Identifiers of the new structures
VALUES = *identifiers*	Values for each new structure
DECIMALS = *scalars*	Number of decimals for printing numerical structures
CHARACTERS = *scalars*	Number of characters for printing texts or labels of a factor
EXTRA = *texts*	Extra text associated with each identifier
MINIMUM = *scalars*	Minimum value for numerical structures
MAXIMUM = *scalars*	Maximum value for numerical structures

The DUPLICATE directive allows you to define new data structures with attributes like those of existing structures. The attributes to be duplicated are defined by the ATTRIBUTES option. The structures from which the attributes are to be taken are specified by the OLDSTRUCTURES parameter, while the structures that are to be defined are specified by the NEWSTRUCTURES parameter. The other parameters allow some of the more important attributes to be reset at the same time. This is illustrated in Example 2.10.3, where the factor Species2 takes its levels (and thus its number of levels) from the factor Species1. However, the labels are not transferred, and other values are defined using the VALUES parameter.

Example 2.10.3

```
FACTOR [LEVELS=!(0,1); LABELS=!T(absent,present); \
  VALUES=0,1,1,0,0,0,1] Species1
DUPLICATE [ATTRIBUTES=levels] Species1; \
  NEWSTRUCTURE=Species2; VALUES=!(1,0,1,1,0,1,0)
PRINT Species1,Species2

 Species1    Species2
   absent      1.0000
  present      0.0000
  present      1.0000
   absent      1.0000
   absent      0.0000
```

```
      absent       1.0000
     present       0.0000
```

You can set option `REDEFINE=yes`, to allow `DUPLICATE` to change the type of any pre-defined new structure, if necessary, to have the same type as the corresponding old structure. Otherwise, `DUPLICATE` will report a fault if the new structure has previously been defined to have a different type.

2.10.4 The PDUPLICATE procedure

PDUPLICATE procedure
Duplicates a pointer, with all its components (R.W. Payne).

No options

Parameters

`OLDPOINTER` = *pointers*	Pointers to duplicate
`NEWPOINTER` = *pointers*	Duplicated pointers

`PDUPLICATE` is useful when you want to duplicate the complete tree of data structures to which a pointer points. So, it duplicates not only the pointer itself, but all the structures to which it points. Also, if any of these structures is itself a pointer, the structures to which that too points will be duplicated.

The pointer to be duplicated is specified by the `OLDPOINTER` parameter, and the duplicated pointer is saved by the `NEWPOINTER` parameter.

2.11 Listing or accessing details of data structures

It can sometimes be difficult to remember all the details of your data structures. For example, in a long interactive session you might forget the identifiers of certain structures, or some of the attributes that you have given them. The `LIST` directive (2.11.1) allows you to list the names and important attributes of your currently available data structures, and the `DUMP` directive (2.11.2) allows you to display all their details. `DUMP` can also display internal information about Genstat but this is useful mainly for those extending Genstat. On other occasions you may need to store and not just display the attributes of structures. This can be done using the `GETATTRIBUTE` directive (2.11.3).

2.11.1 The LIST directive

LIST directive
Lists details of the data structures currently available within Genstat.

Options

`PRINT` = *string tokens*	What to print (`identifier`, `attributes`); default `iden`, `attr`
`CHANNEL` = *identifier*	Channel number of file, or identifier of a text to store output; default current output file
`SYSTEM` = *string token*	Whether to include "system" structures with prefix _ (`yes`, `no`); default `no`
`SCOPE` = *string token*	When used within a procedure, this allows the listing of

	structures in the program that called the procedure (`SCOPE=external`), or in the main program itself (`SCOPE=global`), rather than those within the procedure (`local`, `external`, `global`); default `loca`
`NSTRUCTURES` = *scalar*	Saves the number of structures of the requested types
`SAVE` = *pointer*	Saves a pointer containing the structures of the requested types

Parameter

string tokens	Types of structure to list (`all`, `ASAVE`, `diagonal`, `dummy`, `expression`, `factor`, `formula`, `lrv`, `matrix`, `pointer`, `RSAVE`, `scalar`, `sspm`, `symmetric`, `table`, `text`, `tree`, `TSAVE`, `tsm`, `variate`, `VSAVE`); default `all`

The `LIST` directive provides a quick way of finding out about the data structures available in your program. `LIST` is particularly useful when you are working interactively to remind you about the data structures that you have set up, and the identifiers that you have used. At line 10 of Example 2.11.1, we list the variates that have been set up and in line 11 we list the factors.

The parameter specifies the types of structure that you want to list. By default, all types are listed.

By default `LIST` prints details of relevant attributes, as well as the identifiers, but this can be controlled by setting the `PRINT` option only to `identifier`.

Example 2.11.1

```
   2  VARIATE   [NVALUES=12] Count,Dose
   3  READ      Count,Dose

    Identifier   Minimum      Mean   Maximum     Values   Missing
         Count     2.000     122.2     615.0         12         0     Skew
          Dose     100.0     325.0     600.0         12         0

   7  CALCULATE Logcount = LOG10(Count+1)
   8  VARIATE   [NVALUES=12] Fitted
   9  GROUP     Dose; Dgroup
  10  LIST      variate

  Structures of type VARIATE

  identifier  number of values
       Count                12
        Dose                12
    Logcount                12
      Fitted                12

  11  LIST      factor

  Structures of type FACTOR

  identifier  number of values  number of levels
      Dgroup                12                 4
```

The `SYSTEM` option of `LIST` controls whether structures whose identifiers begin with the underscore character _ are listed; this character is used as a prefix for example for the temporary private structures set up by the menus of Genstat *for Windows*, so their inclusion could be confusing. The `SCOPE` option can be used within a procedure to list the data structures in the program that called the procedure (`SCOPE=external`) or in the outermost part of the program

(SCOPE=global).

The SAVE option can save a pointer containing the structures of the requested types. This is not formed if there are none.

The NSTRUCTURES option can save a scalar storing the number of structures of these types. (So you can check whether a SAVE pointer has been formed by checking whether NSTRUCTURES is greater than zero.)

2.11.2 The DUMP directive

DUMP directive

Prints information about data structures, and internal system information.

Options

PRINT = *string tokens*	What information to print about structures (attributes, values, identifiers, space); default attr
CHANNEL = *identifier*	Channel number of file, or identifier of a text to store output; default current output file
INFORMATION = *string tokens*	What information to print for each structure (brief, full, extended); default brie
TYPE = *string tokens*	Which types of structure to include in addition to those in the parameter list (all, ASAVE, diagonalmatrix, dummy, expression, factor, formula, LRV, matrix, pointer, RSAVE, scalar, SSPM, symmetricmatrix, table, text, tree, TSAVE, TSM, variate, VSAVE); default * i.e. none
SYSTEM = *string token*	Whether to display Genstat system structures (yes, no); default no
UNNAMED = *string token*	Whether to display unnamed structures (yes, no); default no

Parameter

identifiers or *numbers*	Identifier or reference number of a structure whose information is to be printed

The structures for which the information is to be displayed are specified by the parameter of DUMP. The PRINT option indicates what is to be presented: you can ask for just the identifiers, or values and identifiers, or attributes (the identifier is itself an attribute), or for all three. For example, this gives all three for the structures A and B:

```
DUMP [PRINT=attributes,values] A,B
```

Example 2.11.2a

```
   2  VARIATE [VALUES=1...8,*] A
   3  FACTOR [NVALUES=9; LEVELS=!(0,1.2,2.4)] B
   4  DUMP [PRINT=attributes,values] A,B

Dump
====

Identifier      Type  Length   Values Missing  Ref.No.

         A  Variate        9  Present       1     -635
        1.0000       2.0000       3.0000       4.0000       5.0000
        6.0000         7.0000         8.0000              *

          B  Factor          9   Absent        *     -632
 No values
```

There is also a setting, `space`, which provides information about the current use of workspace within Genstat.

If the `CHANNEL` option is set to a scalar, this specifies the output channel to which the information is sent. Alternatively, if you specify the identifier of a text structure, the lines of information will be stored in the text instead of being printed; likewise if you specify the identifier of a structure that has not yet been declared, it will be defined automatically as a text to store the information. If `CHANNEL` is not specified, the information is displayed on the current output channel.

The `INFORMATION` option selects which attributes are presented. The default setting `brief` selects only the most important ones. The setting `full` causes all the attributes to be presented, and the setting `extended` also gives details of the structures associated with listed structures.

Example 2.11.2b

```
   5  DUMP [INFORMATION=extended; PRINT=attributes,values] B

     IDENT VECNO  ATTOR   VALOR TYPE  NVAL NVALUE MODE MVPTR OWNER

         B  -632      *       *    2     9      9    3     *     *
  LEVELS = -633  NLEV   = 3  REFLEV = 1
 No values

            -633      *      19    4     3      3    1     0     *
        0.0000         1.2000         2.4000
```

Some of the attributes may be set to unnamed structures. You can obtain further information about any of these by giving its (negative) reference number (as displayed by `DUMP` when indicating its association with another structure) in the parameter list. This is likely to be useful mainly to advanced users.

The `TYPE` option lets you display, in addition, lists of all structures of a particular type, or of several types. For example, if you had forgotten the identifier of a factor, you could give the statement

```
    DUMP [TYPE=factor; PRINT=identifiers]
```

Example 2.11.2c

```
   6  FACTOR [NVALUES=9; LEVELS=3; VALUES=3(1...3)] F1
   7  & [LEVELS=2; VALUES=(1,2)4,1] F2
   8  DUMP [TYPE=factor; PRINT=identifiers]

Dump
====

 List of structure names
        F2        F1         B
```

This lists all the current factors. When `PRINT=attributes` or `values` (or both), the setting `TYPE=all` provides a list of all named and unnamed structures, except system structures. "`PRINT=identifiers; TYPE=all`" lists only named structures.

The `SYSTEM` option allows all the system structures to be dumped: there are many of these, so it is not a good idea to set this option frivolously.

2.11.3 The GETATTRIBUTE directive

GETATTRIBUTE directive

Accesses attributes of structures.

Option

ATTRIBUTE = *string tokens*	Which attributes to access (nvalues, nlevels, nrows, ncolumns, type {type number}, levels, labels {of a factor or pointer}, nmv, present, identifier, refnumber {structure number}, extra, decimals, characters, minimum, maximum, restriction, mode {integer code 1 - 5 denoting type of values: double real, real, integer, character and word}, maxline {of a text or factor}, rows, columns, classification, margins {of a table}, associatedidentifier {of a table}, unknown {cell of a table}, suffixes {of a pointer}, owner, terms {of an SSPM}, groups {of an SSPM}, weights {of an SSPM}, SSPMauxiliary, SSPrst, tsmmodel, rstat {of an RSAVE}, stype {type as a character string}, referencelevel {of a factor}, drepresentation, unitlabels {of a vector}, iprint, datavariate {of a table}, summarytype {of a table}, percentquantile {of a table of quantiles}, %margin {of a table of percentages}, coding {of a text}); default * i.e. none

Parameters

STRUCTURE = *identifiers*	Structures whose attributes are to be accessed
SAVE = *pointers*	Pointer to store copies of the attributes of each structure; these are labelled by the ATTRIBUTE strings

The GETATTRIBUTE directive allows you to access attributes of each of the structures that are listed with its STRUCTURE parameter. It refers to the list of structures by pointers, which are set up by the SAVE parameter. You must always set the option and both parameters. Thus, in Example 2.11.3a, P is defined to be a pointer with an element for each of the two attributes requested by the ATTRIBUTE option. The first is P['nvalues'], alternatively referred to as P[1], storing the value 4; and the second is P['nmv'], or P[2], storing the value 1.

Example 2.11.3a

```
   2  VARIATE [VALUES=1,2,*,4] X
   3  GETATTRIBUTE [ATTRIBUTE=nvalues,nmv] X; P
   4  PRINT P[]

 P['nvalues']    P['nmv']
            4           1
```

If you request an attribute that is not relevant to a structure, it is omitted from the pointer. Thus for example the nlevels, levels and labels settings are relevant only for factors, and nrows and ncolumns only for matrices. The references to the relevant attributes that you specify are always stored in the order shown in the definition of the ATTRIBUTE option above.

For attributes that are single numbers, the information is copied into an unnamed scalar which is pointed to by the appropriate element of the pointer; if the attribute has not been set, then the corresponding scalar will contains a missing value. For the attributes `stype`, `identifier`, `iprint`, `margins`, `associatedidentifier`, `summarytype` and `tsmmodel`, the corresponding element of the pointer is a text structure containing a single line. For the other attributes, the corresponding element of the pointer stores a reference to the attribute itself. One example is the labels vector of a factor. However, if the factor has no labels vector the corresponding entry of the pointer is set to the missing value. Thus, Example 2.11.3b sets up `P` as a pointer with two values, the first being `Lev` and the second missing.

Example 2.11.3b

```
VARIATE [VALUES=4,8,12] Lev
FACTOR [LEVELS=Lev] F
GETATTRIBUTE [ATTRIBUTE=levels,labels] F; P
DUMP [PRINT=attributes,values] P,Lev

Dump
====

Identifier      Type  Length   Values Missing  Ref.No.
         P  Pointer        2  Present       1     -614
    -698       *

       Lev  Variate        3  Present       0     -698
        4.0000       8.0000       12.0000
```

The setting `type` gives a scalar value indicating the type of structure, by the code:

1	scalar	11	expression
2	factor	12	formula
3	text	13	dummy
4	variate	14	pointer
5	matrix	15	LRV
6	diagonal matrix	16	SSPM
7	symmetric matrix	17	TSM
8	table	18	RSAVE
9	ASAVE	22	tree
10	TSAVE		

You can also obtain the type number of a data structure by using the `TYPE` function. The `GCONSTANTS` function can provide the code for each type of structure, as in the table above.

Alternatively, the `stype` setting supplies the type name in a text structure. As the example below shows, this works not only for the standard Genstat types, such as variates and factors, but also for user-defined types (2.7.4).

Example 2.11.3c

```
STRUCTURE    [NAME='complex number'] 'real','imaginary'; TYPE='scalar'
DECLARE      [TYPE='complex number'] C; VALUES=!p(3,2)
FACTOR       F
TEXT         T
VARIATE      V
GETATTRIBUTE [ATTRIBUTE=stype] C,F,T,V; SAVE=Typec,Typef,Typet,Typev
PRINT        Typec[],Typef[],Typet[],Typev[]


Typec['stype'] Typef['stype'] Typet['stype'] Typev['stype']
complex number         factor           text        variate
```

---

3 Input and output

This chapter describes how to read data values into Genstat and how to print them out. It also looks at some of the more general aspects of input and output, such as the use of files for storing different kinds of information.

As already mentioned (1.1), Genstat statements may be typed in at the keyboard or stored in files and executed as a complete program. Similarly, data can be typed in directly, or read from files that have been prepared in advance, or even from the contents of a Genstat text structure. The simplest method is provided by the `FILEREAD` procedure, which provides the basis of the Read Data from ASCII file menu in Genstat *for Windows*. The Windows implementation also allows a wide range of spreadsheet files to be imported, as well as save-files from many other statistical systems and data bases. The most general facilities are provided by the `READ` directive, which caters for a wide variety of styles and formats, and can rescale and sort the data values as they are read.

`READ`	provides general facilities for reading data from the keyboard, an input file or a Genstat text structure (see Sections 3.1.2 to 3.1.12, and 3.7)
`FILEREAD`	provides a convenient way of reading values into a set of variates, factors and or texts which all have equal lengths; the data values are provided in a rectangular layout, in a separate file (3.1.1)
`TX2VARIATE`	reads values into a variate from a text structure (4.5.3)

Genstat can produce output in either plain-text or a "formatted" style written in either RTF, HTML or LaTeX. The style of an output channel is set when the channel is opened, either by the `OPEN` directive (3.3.1) or by the command used to run Genstat (1.1.2). You can also switch a formatted output channel temporarily into the plain-text style (and back into its formatted style) using the `OUTPUT` directive (3.4.3). Alternatively, in Genstat *for Windows*, this is done using the View menu.

The plain-text style assumes that every character occupies an identical width on the page. This was the situation with the line printers that were originally used for computer output. In more modern environments, such as Microsoft® Windows^{TM}, this can be achieved by using a "non-proportional" font such as Courier. In plain text, columns of output are lined up by inserting space characters. The formatted styles insert tab characters or use tabular modes of output, which are likely to be more convenient if you want to import the output into a wordprocessor, web page or scientific publication. Further details are in Section 3.3.1. In the formatted styles, you can also include "typesetting commands" inside a textual string to generate italic or bold fonts, subscripts or superscripts, and Greek or mathematical symbols (see 1.4.2).

Genstat's analysis commands produce output in formats appropriate to the current style. You can generate your own output by "printing" the contents of data structures into output files (or into text structures) using the `PRINT` directive. Titles in Genstat's standard formats can be printed using the `CAPTION` directive. The `PAGE` directive starts future output at the top of the next page, the `SKIP` directive allows blank lines to be inserted in output files (or lines to be skipped in input files), and the `PLINK` procedure allows you to include graphics in an HTML file. The `DECIMALS` and `MINFIELDWIDTH` procedures allow you to set formats automatically.

`PRINT`	prints data in tabular form to an output file or a text (3.2.1, 3.2.2 and 3.7)
`CAPTION`	prints various types of caption and title (3.2.3)
`PAGE`	moves to the top of the next page of an output file (3.2.4)
`SKIP`	skips lines of input or output files (3.3.3)

`PLINK`	prints a link to a graphics file into an HTML file
`DECIMALS`	sets the number of decimals for a structure, using its round-off (3.2.5)
`MINFIELDWIDTH`	calculates minimum field widths for printing data structures (3.2.6)

You can open and close external files from within your Genstat program. Each file is connected to a *channel* (input, output, backing-store, and so on) through which it is accessed by the Genstat commands that read input or generate output.

`OPEN`	opens files, connects them to Genstat input or output channels and specifies aspects such as the line width and output style (3.3.1)
`CLOSE`	closes files, freeing the channels to which they were attached (3.3.2)
`ENQUIRE`	provides details about external files attached to Genstat (3.3.4)

The channel from which input statements are taken can be changed, as can the channel to which output is sent. It is also possible to send a transcript (or copy) of input and/or output to output files.

`INPUT`	specifies the channel from which subsequent statements should be read (3.4.1)
`RETURN`	returns to the previous input channel (3.4.2)
`OUTPUT`	specifies the channel to which future output should be sent, and allows you to switch between plain-text and formatted styles for channels opened as RTF, HTML or LaTeX (3.4.3)
`COPY`	requests a transcript of subsequent input and/or output (3.4.4)

The values of a data structure, with all its defining information, can be stored in a sub-file of a "backing-store" file (3.5). It can then be retrieved in a later job, without the need to repeat the definitions.

`STORE`	stores data structures in a backing-store file (3.5.3)
`RETRIEVE`	retrieves data structures from a backing-store file (3.5.4)
`CATALOGUE`	displays the contents of a backing-store file (3.5.5)
`MERGE`	copies sub-files of backing-store files into a single file (3.5.6)

The current state of the whole job can also be stored, so that it can be picked up and continued on a later occasion.

`RECORD`	saves the complete details of a job (3.6.1)
`RESUME`	reads and restarts a recorded job (3.6.2)

Genstat *for Windows*, has several additional commands for accessing data from spreadsheets, databases and other systems (3.8). However, these may be unavailable in other implementations.

`EXPORT`	Outputs data structures in foreign file formats, or as plain or comma-delimited text
`IMPORT`	Reads data in a foreign file format, and loads it into Genstat or into a Genstat spreadsheet file
`SPLOAD`	loads a Genstat spreadsheet file
`SPCOMBINE`	combines spreadsheet and data files, without reading them into Genstat

`CSPRO`	reads a data set from a CSPro survey data file and dictionary, loads it into Genstat or puts it into a spreadsheet file
`DBCOMMAND`	runs an SQL command on an ODBC database
`DBEXPORT`	Update an ODBC database table using data from Genstat
`DBIMPORT`	Loads data into Genstat from an ODBC database
`DBINFORMATION`	loads information on the tables and columns in an ODBC database
`DDEEXPORT`	Sends data or commands to a Dynamic Data Exchange server
`DDEIMPORT`	Gets data from a Dynamic Data Exchange (DDE) server
`GRIBIMPORT`	reads data from a GRIB2 meteorological data file, and loads it or converts it to a spreadsheet file
`%CD`	Changes the current directory

Details are available in the on-line help.

3.1 Reading data

Although you can define values for data structures when you declare them, using the `VALUES` option or parameter (2.1.1), it is usually more convenient to read the values - especially with large sets of data. Many data sets consist of vectors (i.e. variates, factors or texts) each with the same numbers of values. The most common representation has the data presented in *parallel* (that is, the values for the first units of all the vectors, then values for their second units, and so on) in a separate file from the Genstat program. Data files like this are often read most conveniently using the `FILEREAD` procedure (3.1.1). The alternative is the `READ` directive (Sections 3.1.2 onwards), which can read data into any Genstat data structure using a wide variety of formats.

3.1.1 The `FILEREAD` procedure

`FILEREAD` procedure

Reads data from a file (P.W. Lane).

Options

`PRINT` = *string tokens*	What output to display (`summary`, `groups`, `comments`, `firstline`); default `summ`, `grou`, `comm`, `firs`
`NAME` = *text*	External name of the data file; no default in batch mode, name is prompted for in interactive mode
`END` = *text*	What string terminates data; default `':'` (the end of file also terminates data for any setting); the setting `END=*` is not allowed
`MISSING` = *text*	What character represents missing values; default `'*'`
`SKIP` = *scalar* or *text*	Number of lines to skip at the start of the file, or string to indicate the record before the first record of data; default 0
`MAXCATEGORY` = *number*	The maximum number of categories for which a structure is defined to be a factor unless otherwise specified by `FGROUPS`; default 10
`COMMENTSYMBOLS` = *text*	What characters to treat as introducing comments if found in the first column at the start of the file; default double-quote character (`"`)

`IMETHOD` = *string token*	How identifiers are to be specified for the data structures to be read (`supply`, `read`, `none`); default `supp`
`ISAVE` = *pointer*	To store the identifiers, whether read or supplied, and to provide suffixed identifiers for data with no specified identifiers
`SEPARATOR` = *text*	What (single) character separates successive values; default is the space character

Parameters

`IDENTIFIER` = *identifiers*	Names for the data structures that are to be read; these are prompted for if this is unset when running interactively with `IMETHOD=supply`; identifiers are redefined if they have been used previously
`FGROUPS` = *string tokens*	Whether to form each data structure into a factor (`check`, `form`, `leave`); default `chec`, which causes `FILEREAD` when running interactively to ask about any structure whose number of distinct values is less than or equal to `MAXCATEGORY`, and when running in batch to define as factors all structures with `MAXCATEGORY` or fewer distinct values (note: for compatibility with earlier releases, `yes` and `no` can be used as synonyms of `form` and `leave`)
`REPRESENTATION` = *string tokens*	What representation to assume for each data structure (`numbers`, `characters`); default unset – representation is determined by whether the first value is a number; if set for one structure, this parameter must be set for all structures

`FILEREAD` reads data from a file into variates, factors or texts. It can deal with data values laid out in the following ways.

1) A character file: that is, a normal readable file, or flat file.
2) Maximum record length of 200 characters.
3) Contents consist of values for one or more data structures – usually presented as a single rectangular data matrix.
4) The values for the data structures are recorded in parallel – that is, the first values of all the structures, followed by the second values of all, and so on; usually, each record of the file contains one value of each structure, but multiple values per record and multiple records for each unit can also be dealt with.
5) Values in a record are separated from each other by the same separator – usually one or more spaces.
6) Text values must be enclosed in single quotes if they contain a space, comma, backslash or double-quote; single-quotes must be used only to enclose textual values, or be duplicated as part of a value which is also enclosed in single quotes.
7) Comments are allowed at the start of the file only if every record to be treated as a comment starts with a double quote or other specified symbol. Alternatively, a specified number of records at the start of the file can be skipped, or any number of records up to and including a specified string.
8) Identifiers for the columns of the matrix can be read from the first row of data, as long as they are valid, unsuffixed, Genstat identifiers. An exclamation mark after an identifier signals that the structure is to be set up as a factor.

The information in each data structure can be either be numerical or textual. FILEREAD can usually discover which is appropriate for each structure automatically, by examining the data. Alternatively, you can specify this explicitly, using the REPRESENTATION parameter. If REPRESENTATION is unset, FILEREAD looks at the first record in the file with no missing values to see whether the value provided for each structure can be interpreted as a number (if not, it is taken to be a textual string), and FILEREAD will fail if there is no such record. If the REPRESENTATION parameter is set for any structure, it must be set for all of them.

The NAME option supplies the name of the file, enclosed in single quotes. If you are running Genstat in batch mode the name must be supplied but, in interactive mode, FILEREAD will prompt you for the name if the NAME option is unset.

The IMETHOD option controls how the identifiers are specified for the structures to be read. With the default, IMETHOD=supply, the identifiers can be listed using the IDENTIFIER parameter, one for each column of the data matrix. If IDENTIFIER is not set when running Genstat interactively, FILEREAD will prompt for identifiers; if it is unset when running in batch, FILEREAD just reports on the contents of the file, unless option ISAVE is set (see below). If IMETHOD=read, FILEREAD will read the identifiers for the data structures from the first complete record in the file (and the IDENTIFIER parameter is then ignored). They must be valid Genstat identifiers, and must not include suffixes. If an exclamation mark is found after (or in) an identifier, the structure will be set up as a factor unless the FGROUPS parameter is set to leave for that structure (see below). If IMETHOD=none, FILEREAD just reports on the contents of the file without assigning identifiers, unless option ISAVE is set.

The ISAVE option can be set to a pointer to store the identifiers read from the file (if IMETHOD=read) or supplied interactively (if IMETHOD=supply). Alternatively, if ISAVE is set and no identifiers are specified (that is, if IMETHOD=none when running either interactively or in batch, or if IMETHOD=supply and the IDENTIFIER parameter is unset when running in batch), the data will be read into suffixed identifiers of the ISAVE pointer.

Values on the same record of a file must be separated from each other by at least one space unless the SEPARATOR option is set. This option can nominate any single character to be treated as data separator. The MISSING and END options specify symbols to represent the missing-value and to denote the end of the file.

If the number of identifiers is not specified, the number of data structures is taken to be the number of values on the first record with no missing values. But if identifiers are supplied using the IDENTIFIER parameter, or are read from the data file, it is possible to read several units of data from each record or each unit from several records. If there are more values on the first record of data than there are identifiers, the type of each data structure can be determined only by its first value: FILEREAD will fail if any first value is missing, unless the REPRESENTATION parameter is set. If there are fewer values on the first record of data than there are identifiers, FILEREAD will fail regardless of the absence of missing values unless the REPRESENTATION parameter is set.

The PRINT option controls the various reports produced by FILEREAD, according to the following settings.

summary	names, types, numbers of values and missing values of the structures
groups	table of the number of values in each category for structures that have *n* or fewer distinct values (where *n* is the limit specified by the MAXCATEGORY option)
comments	any comments found before the start of the data
firstrecord	the first record of data that contained no missing values

By default all four reports are produced.

The FGROUPS parameter allows structures to be turned automatically into factors. The default setting is check: when running Genstat interactively, FILEREAD will then prompt you for a decision about any structure where the number of distinct values is less than or equal to the setting of the `MAXCATEGORY` option; in batch, all structures with these few distinct values become factors automatically. `FGROUPS` can also be set to `form` or `leave` to specify explicitly whether each structure should or should not be defined automatically as a factor. (The settings `form` or `leave` were introduced in Procedure Library PL21 to replace the settings `yes` and `no`, as other options and parameters that have no as a setting, use `no` as their default. However, for compatibility with earlier programs, the settings `yes` and `no` are currently still recognized as synonyms for `form` and `leave`.)

The `COMMENTSYMBOLS` option can be set to a list of single characters, in quotes. If any of these characters is found at the start of a record, before any data value has been read, that record will be treated as a comment. By default, the double-quote symbol is the only comment symbol, but it must appear at the start of every record to be treated as a comment.

The `SKIP` option allows records at the start of the file to be skipped altogether. It can be set either to the number of records to be skipped, or to a string, indicating that all records are to be skipped up to and including the first record containing that string.

3.1.2 Introduction to the `READ` directive

When you use `READ`, you can type the data values at the keyboard, read them from the file containing your Genstat program, or read them from a separate data file. The following simple example shows how to read the values for a variate called `Weights`:

```
VARIATE [NVALUES=10] Weights
READ Weights
24.3 25.6 57.3 43.8 45.3
46.5 47.9 97.0 77.5 64.3 :
```

There are many options and parameters to allow control over most aspects of data input, so data can be read in almost any form. We first describe the more straightforward uses of `READ`, with most of the options retaining their default settings. Then, in 3.1.3, we give the full details of the syntax of `READ` showing how you can use the options and parameters to read data that may be arranged in many other ways.

Unless specified otherwise, Genstat assumes that the data values will be found immediately after the `READ` statement. The values are usually specified in *free format*: that is, they are separated by one or more spaces (or tabs) and can be arranged any way you like, on one or more lines, so long as the correct order is maintained. Genstat reads the data one line at a time, so the first element of `Weights` is 24.3, the second element is 25.6, and so on. There is no need to use the continuation character \ when data for `READ` is spread over several lines; in fact \ should occur only when it is part of a string that is being read into a text. To show that the end of the data has been reached a *terminator* is needed, which by default is a colon (:). This may be at the end of the last line of data or on a line of its own. Once the terminator has been read a simple summary of the data is printed and a quick examination can indicate if `READ` was successful, or if there were any problems such as incorrectly typed values.

Example 3.1.2a

```
  2  VARIATE [NVALUES=10] Weights
  3  READ Weights

   Identifier   Minimum      Mean   Maximum    Values   Missing
      Weights     24.30     52.95     97.00        10         0
```

If the minimum value of `Weights` was less than zero, you might assume there was some kind of problem with the data!

When you are working interactively, Genstat produces a prompt indicating the name of the data structure and the unit number of the next value it expects to read:

Example 3.1.2b

```
> VARIATE [NVALUES=10] Weights
> READ Weights
Weights/1> 24.3 25.6 57.3
Weights/4> 43.8 45.3 46.5
Weights/7> 47.9 97.0 77.5 64.3
* MESSAGE: You have input sufficient data, READ terminated.

    Identifier   Minimum      Mean   Maximum    Values   Missing
       Weights     24.30     52.95     97.00        10         0
>
```

READ prompts for the first data value, Weights/1>, and the first three values are typed in. The next prompt is Weights/4>, requesting values for the fourth and subsequent units of Weights. Because Weights was declared to have 10 values, Genstat will know to stop reading data once 10 values have been typed in. Once the 10th value (64.3) has been typed and the <RETURN> key pressed, READ automatically terminates input, without asking for the terminating colon, although it is quite correct to include it at the end of the last line of data. If you type too many values by mistake you will get a warning message telling you that the extra data has been ignored.

When running Genstat in batch, unless you set the END option (3.1.3), you must mark the end of the data with a colon; READ then checks that you have given the correct number of values. If there are too few values a warning is printed and the data structure is completed by using missing values, whereas a fault will be produced if there are too many values.

Genstat will also perform range checks when reading data if you have set the MINIMUM or MAXIMUM parameters when declaring data structures.

Note that, whether you are running Genstat interactively or in batch, READ will immediately take a fresh line of input, so the data cannot be on the same line as the READ statement; also any characters after the terminating colon will be ignored.

Any numerical structure can be read in this way: scalars, variates, matrices, symmetric and diagonal matrices and tables. The values can be entered in any of the forms described in 1.5.1, that is,

```
1.20  -.2   3e1  -1.25E-2  27  *
```

are all valid, with * indicating a missing value.

The values for rectangular and symmetric matrices and multi-way tables must be given in the order described in 2.4 and 2.5. The rules for free format allow you to arrange them in any way you like, as long as you maintain the correct order, but you will probably find data files easier to manage if the layout corresponds to the dimensions of the data structure: for example

```
SYMMETRIC [NROWS=10] Galaxy
READ Galaxy
0
1.87 0
2.24 0.91 0
4.03 2.05 1.51 0
4.09 1.74 1.59 0.68 0
5.38 3.41 3.15 1.86 1.27 0
7.03 3.85 3.24 2.25 1.89 2.02 0
6.02 4.85 4.11 3.00 2.11 1.71 1.45 0
6.88 5.70 5.12 3.72 3.01 2.97 1.75 1.13 0
4.12 3.77 3.86 3.93 3.27 3.77 3.52 2.79 3.29 0  :
```

Note, however, that the shortcuts for compacting number lists described in 1.5.1 are not allowed within READ. All the values must be given; that is, pre- and post- multipliers and progressions are not recognized. Thus in some cases it will be easier to assign values when declaring your data structures. For example,

```
VARIATE [VALUES=1...365] Day
```

is simpler to type than

```
READ Day
```

followed by 365 individual values.

Textual values (strings) must be enclosed within single quotes if they contain any characters that have special meaning to `READ` (space, tab, comma, colon, asterisk, backslash, single or double quote). The quotes can be omitted for other strings. For example:

```
TEXT [NVALUES=5] Country
READ Country
Australia  Canada  'Great Britain'  U.S.A.  'New Zealand' :
```

The rules for strings in `READ` are thus slightly different to those for lists of strings (1.5.2), where quotes are required for any string that does not start with a letter or contains any character other than letters or digits. Thus `Newcastle-on-Tyne` and `500Km` are both valid when read in as data, but not in a `TEXT` declaration.

Factors can be read using either their numeric levels or the associated textual labels (but you cannot use both methods for the same factor within a single `READ` statement). You can also let `READ` set up the factor levels or labels according to the values that it finds when reading the data (Example 3.1.2d).

If you want to read the values of a pointer (that is, a list of identifier names) the rules are rather stricter than for other types of data, as explained in 3.1.5.

You cannot read formulae or expressions directly. The easiest way to do this is to read the required value into a text which can then be used in an appropriate declaration using either the macro-substitution symbols `##` (1.6.2) or the `EXECUTE` directive (5.4.3). You cannot read values into the compound data structures described in 2.7 (SSPMs, LRVs and TSMs); these should be formed using the appropriate directives (`FSSPM`, `FLRV`, `FTSM`), or by reading the individual components of these structures.

You can read values for more than one structure in a single `READ` statement. The values can be taken either *serially* or in *parallel*. The default is to take the values in parallel: the first element of each structure is read, then the second element of each, until all the data are read. For example:

$a_1\ b_1\ c_1$
$a_2\ b_2\ c_2$
$a_3\ b_3\ c_3$
$a_4\ b_4\ c_4$:

or

$a_1\ b_1\ c_1\ a_2$
$b_2\ c_2$
$a_3\ b_3\ c_3\ a_4\ b_4\ c_4$:

Here `A`, `B` and `C` are in parallel, each with four values. The complete set of values for all three structures is given, followed by one terminating colon. The term *parallel* merely indicates the order in which `READ` is to read the values: that is, the first element of each structure, then the second element of each, and so on. It is not necessary for the data to be laid out in neat columns, although this may make a data file easier to work with.

Different types of structures can be read in parallel and they may have different kinds of values (numerical or text), as shown in Example 3.1.2c.

Example 3.1.2c

```
> VARIATE [NVALUES=5] Area
> TEXT [NVALUES=5] Country
> READ Country,Area
Country/1> Australia 2975.0 Bolivia 424.18 Canada
   Area/3> 3851.9 Denmark 16.618 Ethiopia 457.28 :
```

Notice how the name of the first country is followed by its area, then the name of the second country and so on. Working interactively, the prompt helps you to keep track of which values you need to type next. If you want to read data in parallel, all the data structures must be the same length.

When reading in serial mode, all the values of the first structure are read, then all the values for the second structure, until all the data structures have been read. For example

x_1 x_2 x_3 :
y_1 y_2 :
z_1 z_2 z_3 z_4 z_5 z_6 :

Here all the values of `X` are given first, followed by all the values for `Y`, and then all the values for `Z`. Unlike the parallel layout, each set of values must end with the terminating colon, so that `READ` can tell when to move on to the next structure; this means that the structures can be of different lengths.

In all the examples so far we have defined the type and size of the data structures in advance of the `READ` statement. However, `READ` can make some declarations and definitions by default. Any identifier that has not been declared previously will be set up as a variate. Vectors (variates, texts and factors) of previously unspecified size will be set up to the current units length, if set by `UNITS` (2.3.4); otherwise `READ` sets their length to match the number of values read. Also, factors can be generated automatically from the values found, with `LEVELS` or `LABELS` set up as appropriate. The exact rules are described below (3.1.4) but a simple illustration, in Example 3.1.2d, shows how to use `READ` to set labels (for factor `Location`) and levels (for factor `Year`).

Example 3.1.2d

```
FACTOR Location,Year
READ [PRINT=data,errors,summary] Location,Year; \
  FREPRESENTATION=labels,levels
England 1979  Australia 1979  Netherlands 1981  France 1983
England 1985  Italy 1987  Australia 1988  Scotland 1989
Netherlands 1991 'New Zealand' 1992  Canada 1993  England 1993
Australia 1994  Ireland 1995  Australia 1996  England 1997
Australia 1999  Poland 1999  Australia 2001  England 2001 :

 Identifier     Values   Missing     Levels
   Location         20         0         10
       Year         20         0         16
```

You can also use option `SETNVALUES=yes` to ensure that any previous setting of length of a vector is reset according to the numbers of values in the new data. This option can be used only when reading variates, texts or factors; more complex structures such as matrices and tables must be declared in advance.

With small amounts of data it may be convenient to type it in directly, or to include it within your Genstat program. However, when you analyse larger data sets it may be more convenient to read the data from a separate file. The use of different files and input channels is explained in full in 3.3, but to use data files with `READ` only the simpler features are required. All you need do is open the data file on another input channel and then tell Genstat to read from that channel. Suppose, for example, the data are stored in a file called `Weights.dat`:

```
24.3 25.6 57.3 43.8 45.3
46.5 47.9 97.0 77.5 64.3 :
```

You need to decide which input channel to use (here channel 4), and then set `CHANNEL` appropriately in `OPEN` and `READ`:

```
OPEN 'Weights.dat'; CHANNEL=4; FILETYPE=input
READ [CHANNEL=4] Weights
```

The data file is just an ordinary text file, which may have been created within an editor or data-entry system, or perhaps as output from another program. You can still use the other options of

READ, to read multiple data structures in serial or parallel format and so on. You may need to edit the file, for example to insert a colon after each set of data, or you can use the facilities described in 3.1.3 for reading data sets that do not meet the default rules.

We now explain the various options and parameters of READ in more detail and introduce some other ways in which data may be read: in *fixed format*, or from *unformatted* (binary) files, or from Genstat text structures. There are options available to make it easier to read very large amounts of data and to skip over unwanted sections of data. Also you can specify your own characters or strings to separate data values, indicate missing values and mark the end of data.

3.1.3 Syntax of the READ directive

READ directive

Reads data from an input file, an unformatted file or a text.

Options

PRINT = *string tokens*	What to print (data, errors, summary); default erro, summ
CHANNEL = *identifier*	Channel number of file, or text structure from which to read data; default current file
SERIAL = *string token*	Whether structures are in serial order, i.e. all values of the first structure, then all of the second, and so on (yes, no); default no, i.e. values in parallel
SETNVALUES = *string token*	Whether to set number of values of vectors from the number of values read (yes, no); default no causes the number of values to be set only for structures whose lengths are not defined already (e.g. by declaration or by UNITS)
LAYOUT = *string token*	How values are presented (separated, fixedfield); default sepa
END = *text*	What string terminates data (* means there is no terminator); default ':'
SEQUENTIAL = *scalar*	To store the number of units read (negative if terminator is met); default *
ADD = *string token*	Whether to add values to existing values (yes, no); default no (available only in serial read)
MISSING = *text*	What character represents missing values; default '*'
SKIP = *scalar*	Number of characters (LAYOUT=fixe) or values (LAYOUT=sepa) to be skipped between units (* means skip to next record); default 0 (available only in parallel read)
BLANK = *string token*	Interpretation of blank fields with LAYOUT=fixe (missing, zero, error); default miss
JUSTIFIED = *string tokens*	How values are to be assumed justified with LAYOUT=fixe (left, right); default righ
ERRORS = *scalar*	How many errors to allow in the data before reporting a fault rather than a warning, a negative setting, -*n*, causes reading of data to stop after the *n*th error; default 0
FORMAT = *variate*	Allows a format to be specified for situations where the layout varies for different units, option SKIP and parameters FIELDWIDTH and SKIP are then ignored (in the variate: 0 switches to fixed format; 0.1, 0.2, 0.3 or

	0.4 to free format with space, comma, colon or semi-colon respectively as separators; `*` skips to the beginning of the next line; in fixed format, a positive integer *n* indicates an item in a field width of *n*, `-`*n* skips *n* characters; in free format, *n* indicates *n* items, `-`*n* skips *n* items); default `*`
`QUIT` = *scalar*	Channel number of file to return to after a fatal error; default * i.e. current input file
`UNFORMATTED` = *string token*	Whether file is unformatted (`yes`, `no`); default `no`
`REWIND` = *string token*	Whether to rewind the file before reading (`yes`, `no`); default `no`
`SEPARATOR` = *text*	Text containing the (single) character to be used in free format; default `' '`
`SETLEVELS` = *string token*	Whether to define factor levels or labels (according to the setting of `FREPRESENTATION`) automatically from those that occur in the data (`yes`, `no`); default `no` causes them to be set only when they are not defined already
`TRUNCATE` = *string tokens*	Truncation of leading or trailing spaces of strings read in fixed format (`leading`, `trailing`); default `*` i.e. none
`CASE` = *string token*	Whether the case of letters (small and capital) should be regarded as significant or ignored when forming factor labels automatically (`significant`, `ignored`); default `sign`
`LDIRECTION` = *string token*	How to define the ordering of levels or labels when these are formed automatically (`ascending`, `given`); default `asce`

Parameters

`STRUCTURE` = *identifiers*	Structures into which to read the data
`FIELDWIDTH` = *scalars*	Field width from which to read values of each structure (`LAYOUT=fixe` only)
`DECIMALS` = *scalars*	Number of decimal places for numerical data containing no decimal points
`SKIP` = *scalars*	Number of values (`LAYOUT=sepa`) or characters (`LAYOUT=fixe`) to skip before reading a value
`FREPRESENTATION` = *string tokens*	How factor values are represented (`labels`, `levels`, `ordinals`); default `leve`

The `PRINT` option has three settings, `data`, `errors` and `summary`, which control printed output from the `READ` directive. The default is `PRINT=errors,summary`. This produces a printed summary of the data that has been read, and asks for warning messages to be printed about any errors in the data (such as an incorrect number of values); 3.1.13 explains what happens after errors have occurred. The setting `data` will print a copy of each line of input as it is read; this may be useful if data are being read from a file, especially if there are errors. If you set `PRINT=*` no output is produced; you should do this only if you are sure there are no errors in the data.

For numerical structures the printed summary includes the message `Skew` if the values have a markedly skew distribution; that is, if the difference between mean and minimum is more than three times, or less than a third of, the difference between maximum and mean. The summaries can be useful as a quick check that the data have been read successfully, and do not contain any gross errors such as a mistyped number with the decimal point in the wrong position. A separate summary is produced for factors which indicates how many levels are defined for each; you can use this to check that READ has defined the factors correctly when the option SETLEVELS=yes has been set. The summary also indicates the number of missing values read into each structure; these may affect the results of subsequent analyses.

By default, READ will expect to find the data on the current input channel. Working interactively this is the terminal, so a prompt is produced indicating that data is required. When Genstat is being run in batch, the data should start on the line following the READ statement. If you want to read data from another file it should first be opened on another input channel (3.3.1), then the CHANNEL option should be set to that channel number. You can also use CHANNEL to read from a text structure (3.1.9), and by setting UNFORMATTED=yes you can read from an unformatted binary file. In the last case, CHANNEL will refer to a file opened specifically for unformatted access; this is discussed separately in 3.7. Note: you should use CHANNEL if you want to use READ in a program-control structure (5.2) or in a procedure (5.3).

If you specify more than one structure to be read, it is assumed that you want to read the data in parallel. If you want to read in the structures one at a time (for example when they are of different lengths) you should set the option SERIAL=yes.

The default terminator for marking the end of data is the colon (:) but you can use the END option to change this to any string of up to eight characters, for example ENDDATA. If you have defined the size of data structures in advance you can set END=* to indicate that there is no terminator; Genstat then just reads the required number of values. You can omit the terminator from the data if it is stored at the end of a file as the read will be terminated by the end-of-file marker; end-of-file will always terminate the data, whatever the setting of END.

By default, a missing value should be indicated by an asterisk (*); this means that any data item that begins with * is treated as missing. For example, any of the three strings

```
    *    ***    *789
```

will be treated as missing. You can use the MISSING option to change this to any other single character; for example, if you set MISSING='-' then any negative numbers will be read as missing values.

In free format, values are usually separated by spaces or tabs. The SEPARATOR option can be used to specify another character to use as a separator. For example you can use a comma:

```
    READ [SEPARATOR=','] Weights
    24.3, 25.6, 57.3, 43.8, 45.3,
    46.5, 47.9, 97.0, 77.5, 64.3 :
```

You can use spaces and tabs in addition to the specified separator, so long as the separator is present between each pair of values (except at the end of line, when it may be omitted).

The SEPARATOR, END and MISSING strings are all case-sensitive; for example, END=enddata is different from END=EndData. The missing-value and separator characters must be distinct and neither may be part of the END string. This is so that READ can make sense of the input data.

A file can contain several sets of data: for example, it might contain 50 measurements on heights of plants, followed by 50 values of weights. You could read the first 50 by one statement, and the next 50 by another. Genstat maintains a pointer to the current position in each input channel, and so returns to the correct place for the second READ (note that if the first READ finished part-way through a line of data the next READ will start at the next line of the data file). Occasionally you may want to go right back to the beginning of the file; you can do this by setting the REWIND option to yes. For example, if you are working interactively and make a mistake in READ so that the data in a file is read incorrectly, it may be easiest to start all over again with a new READ statement rewinding to the beginning of the file.

Although READ is probably easiest to use when the data are in free format, you may sometimes need to read data using a fixed format. This is selected by the option setting LAYOUT=fixed, described in 3.1.7. You can use the options BLANK, JUSTIFIED and TRUNCATE and parameters FIELDWIDTH and DECIMALS to control reading in fixed format. Alternatively, the FORMAT option caters for more complex examples of free-format or fixed-format data and also allows you to switch between the methods whilst reading; this is discussed in greater detail in 3.1.8.

3.1.4 Implicit declaration of structures

READ can define some of the properties of vectors automatically from the values that are read. More complicated data structures, such as matrices or tables, must be fully defined in advance; for the remainder of this section it is assumed that you are reading vectors.

If the structures to be read have not previously been declared, they will be set up to be variates. If you have already used the UNITS directive (2.3.4) to define a default length then this will apply to any vectors of unknown length in READ. When the structures are being read in parallel (that is, according to the default setting SERIAL=no), they must all be the same length; any vectors of unknown size will be set to the same length as the other vectors being read. If none of the structures has a previously defined length, then READ will act as if SETNVALUES had been set to yes, so that vectors will have their lengths defined from the number of data values found. When reading serially (SERIAL=yes), the structures are treated individually, and any structure of unknown length will be defined from the number of values read in, as if you had set SETNVALUES=yes. You can of course also set SETNVALUES=yes explicitly, to ensure that vector lengths are set from the data, even when they had previously been set to a different size. If you use SETNVALUES when reading structures in parallel with the units vector, slightly different rules apply (3.1.12).

The following examples illustrate some of these rules. X and Y are assumed to be undeclared previously, unless otherwise shown:

```
VARIATE [NVALUES=5] X
READ X,Y
```

declares Y to be a variate of length 5 (like X);

```
VARIATE [NVALUES=5] X
READ [SERIAL=yes] X,Y
```

expects five values for X, and defines Y as a variate with its length defined from the number of values found in the second set of data;

```
READ X,Y
```

defines X and Y as variates of the same length, calculated from the number of values found (which must be a multiple of 2);

```
READ [SERIAL=yes] X,Y
```

defines X and Y from the number of values found, which may be different for each variate.

You can also let READ define the levels and labels of factors automatically. If you just define an identifier to be a factor, and do not mention either levels or labels, READ can set these from the values that are read. The FREPRESENTATION parameter (3.1.5) controls how this is done. If FREPRESENTATION is set to ordinals, the values should all be positive integers, and the number of levels is set equal to the largest number that is read. With the default setting, levels, the values can be any real numbers; the levels of the factor are formed from all the distinct values in the data. Similarly, with FREPRESENTATION=labels, the factor values are supplied as strings, and Genstat forms factor labels from the different strings that are found. By default READ distinguishes between capital and small letters when forming factor labels, but you can set option CASE=ignored to ignore the case of letters. Also, by default the levels or labels are sorted into ascending order, but you can set option LDIRECTION=given to leave them in the order in which they are found in the data file. This may be useful for example if you are reading compass directions or days or months. (With levels or labels, the method is the same as that used by the GROUPS directive (4.6.1) when neither the NGROUPS option nor the LIMITS parameter are set: you could obtain the same factors by reading a variate or text and then using GROUPS to form the factor yourself.)

You can use the option SETLEVELS=yes to force the definition of factors in READ so that any previous labels or levels are overwritten. The lengths of factors can also be set by READ, according to the rules already defined. For example,

```
    FACTOR [NVALUES=5] Age
    READ [SETLEVELS=yes] Age; FREPRESENTATION=ordinals
    21 22 21 24 29 :
```

sets up the factor AGE with 29 levels, that is, as if the FACTOR statement had the option setting LEVELS=29. In contrast, the setting FREPRESENTATION=levels would form Age as a factor with the four levels (21,22,24,29).

If you have defined your data structures in advance, READ implicitly includes a check on the validity of your data: that it has the correct number of values and, when reading factors, that the correct values are given. Although it may be more convenient to let READ set up your data structures, you need to be careful as there is then no longer any check on the input values. It is unwise to suppress the printed summary (3.1.1); this will tell you how many values have been read, how many levels have been set up for factors, and so on. One point to remember is that, if READ is defining the levels or labels of factors automatically, any misspelt value will generate an unwanted level. If there seem to be too many levels, you might want to use TABULATE afterwards to print the levels with their replications (4.11.1).

3.1.5 Reading non-numerical data: texts, factors and pointers

The rules for the interpretation of strings in READ are different from those when string lists occur in a statement (1.5.2). Double quotes and backslashes are accepted as ordinary characters, and the strings cannot be continued over a line.

In free format, quotes are required around any string that contains a space, quote, tab, colon, asterisk, backslash, double quote, or character specified in the SEPARATOR or MISSING options. If you have set END, the end string would need to be quoted if you also wanted to read it as a data value. In a quoted string, any of the aforementioned characters are treated literally, except for the single quote which must be repeated. A textual missing value can be represented by either a quoted empty string (''), or the missing value character (*, unless set otherwise by the MISSING option). An asterisk (or any other character representing the missing value) can still be read, provided it is put within quotes: '*'.

```
    TEXT Heading
    READ Heading
    '*** Latent Roots of X''X ***':
```

The value stored in Heading is `*** Latent Roots of X'X ***`.

The values of factors are usually represented by their levels. You can change this by setting the FREPRESENTATION parameter. If you set it to labels, READ will accept as values the labels of the factor, using the rules for reading text described above. The strings given as data values must exactly match the labels of the factor if they have been declared. The setting FREPRESENTATION=ordinals causes READ to expect an integer in the range 1 up to *n*, the number of levels declared for the factor. As FREPRESENTATION is a parameter it can be set to a list of values which are cycled in parallel with the structures to be read. Thus, you are allowed to read several factors in one READ statement, possibly using a different method for reading each one. The setting of this parameter is ignored for any structures that are not factors, but remember that the list will still be cycled in parallel with these other structures.

The values of pointers are identifiers, that is, names of other data structures. When reading a pointer only simple identifiers are allowed: suffixes cannot be used. For example, Winston is allowed but Orwell[1984] is not.

The rules for reading text and factor labels are slightly different if you are using fixed format (LAYOUT=fixed). These is explained at the end of 3.1.7.

3.1.6 Skipping unwanted data (in free format)

You may sometimes have a data file that contains more data than you want to read in to Genstat. For example, there may be several lines at the beginning of a file to describe the data set. You can use the SKIP directive (3.3.3) to skip over these lines before using READ to read in the actual data. Alternatively, you can embed the description in double-quotes (") and make it into a comment that READ will ignore. You can also use comments to annotate your data or to remove some values temporarily from the data.

If you want to skip over some of the data systematically, as for example when there are several columns and only some are required for your analysis, there is an option and a parameter that you can use, both of which are called SKIP.

The SKIP option indicates how many values to skip between complete units of data. For example, with a file in channel 2 containing five columns of data, the statement

```
READ [CHANNEL=2; SKIP=3] X,Y
```

would read X and Y from the first two columns, and then skip the final three columns: Genstat reads the first value for X and Y, the next three values are skipped before reading the second value of X; so READ moves onto the next line of the file, and so on. You can also set SKIP=* to skip directly to the next line of data; you could use this if there were varying numbers of additional columns in the file. By default, SKIP is zero, so no values are skipped.

The SKIP parameter is interpreted in parallel with the structures whose values are to be read. It indicates how many values should be skipped before reading the value for the corresponding structure. This is easiest to explain in terms of parallel columns (although the rules for free format do allow other actual layouts of the data).

```
31 91 11 81 21
32 92 12 82 22
33 93 13 83 23
34 94 14 84 24
35 95 15 85 25:
```

To read only the first, third and fifth columns, we could type

```
READ A,C,E; SKIP=0,1,1
```

The SKIP parameter tells Genstat to skip no values before reading A and one value before reading C and reading E. Thus Genstat reads the values shown in bold. This statement would work in exactly the same way if the data had been laid out differently: for example

```
31 91 11 81 21 32 92 12 82 22 33 93 13 83 23
34 94 14 84 24 35 95 15 85 25:
```

The SKIP option can be used in conjunction with the parameter when additional values need to be skipped between units of data. In the example above, to skip over the values shown in bold and read the intervening columns instead, the statement

```
READ [SKIP=1] B,D; SKIP=1
```

could be used with either layout of values. With the parallel layout of data, setting option SKIP=* would work equally well, but this would not work with the data in the more compressed layout.

The FORMAT option (3.1.8) also allows you to skip unwanted values or lines of data, but is most useful when the data file contains more complex arrangements of data. If you set FORMAT, the SKIP option and parameter will be ignored. In fixed format data is skipped one character at a time, rather than one value at a time; this is described in the next section.

3.1.7 Reading fixed-format data

In fixed format, data values are arranged in specific *fields* on each line of the file. Each field consists of a fixed number of characters. There is no need for separating spaces; the tab character is not permitted, nor are comments. So, depending on how the fields are defined, the sequence of digits `123456` could be interpreted for example as the single number 123456, or two numbers 123 and 456, or three numbers 123, 4 and 56. Data like this are usually produced by special-purpose programs or equipment; for example, automatic data recorders.

To read data in fixed format you set the `LAYOUT` option to `fixed`, and then specify the format to be used. If the values for a structure always occupy the same number of character positions, you can do this with the `FIELDWIDTH` parameter. For example,

```
    READ [CHANNEL=2; LAYOUT=fixed] Weight,Height; FIELDWIDTH=3,5
```

takes data from channel 2 in fixed format. The data are in parallel: that is, reading across lines of the file, values for `Weight` and `Height` appear alternately. The `FIELDWIDTH` parameter is processed in parallel with the structures to be read, so each item of `Weight` data takes up three characters, and each item of `Height` data takes up five. If the fieldwidth for a structure is not constant, that is if different layouts are used for different units of the data, then you need to use the `FORMAT` option, described in the next section (3.1.8).

Suppose there are 80 characters per line in the file; each pair of `Weight` and `Height` values takes up 8, and so you have 10 pairs per line. The first line looks like:

Weight$_1$Height$_1$Weight$_2$Height$_2$ ... Weight$_{10}$Height$_{10}$

Suppose that the first two values for Weight were 1 and 200, and that the first two for Height were 10 and 1200. Then, using ␣ to represent a space, the first four items on this line would be:

```
    ␣␣1␣␣␣10200␣1200
```

Genstat is able to identify the separate values 10 and 200 because it is reading a fixed number of characters for each structure.

Genstat input files have a nominal width, set by default to 80. This can be altered by an `OPEN` statement (3.3.1) to a different value if necessary. When reading in fixed format, each line of input is taken to be exactly this width; shorter lines are extended with spaces (blanks). It is important to make sure that you account for this when setting the options for `READ`, otherwise you may read some values from these blank fields (the `BLANK` option, described below, explains how the blank fields would be interpreted). In the example above, if the values for `Height` occupied four characters instead of five there would be 11 pairs of values per line of 77 characters. Using the default settings, the final three characters on the first line would be read as the 12th value of `Weight`, and `READ` would then be out of step as the 12th value of `Height` would be read in from the beginning of the next line. The simplest solution is to set the file width to 77 in the `OPEN` statement, but you can also use the `SKIP` option and parameter (see below) or the `FORMAT` option (3.1.8) to avoid this sort of problem.

When you are using fixed format, the data terminator must begin within the first field to be read after the final data value: so you must ensure that you set the field widths and position the terminator appropriately. If you are using either the `SKIP` option or parameter, you must take care not to skip accidentally over the terminator, as `READ` will continue to take input - and probably generate many error messages.

Normally Genstat treats a blank field in fixed-format data as a missing value, and the only indication will be in the count of missing values in the printed summary. You can request warning messages for blank fields by setting the option `BLANK=error`. Alternatively, you can cause blanks to be interpreted as zeroes, by setting `BLANK=zero`.

Data in fixed format are normally taken to be right-justified: that is, their right-hand ends are flush with the right-hand end of the field; you can have either blanks or leading zeroes (for numbers) in the redundant spaces at the left of the field. You can change this default by setting the `JUSTIFIED` option. For example the value 123 can appear in a field of width 5 as:

`␣␣123`	`JUSTIFIED=right`	there may be leading blanks (the default)
`123␣␣`	`JUSTIFIED=left`	there may be trailing blanks
`00123`	`JUSTIFIED=left,right`	there must be no blanks
`␣123␣`	`JUSTIFIED=*`	there may be leading or trailing blanks

In this way, `JUSTIFIED` allows you to check the blanks in each field. If a data field contains any blanks that are not allowed by the current setting, an error will be reported. Note that when reading numerical data, embedded blanks are never permitted. So a field containing, for example `1␣2␣3`, will always produce an error message.

As an example, we can read the values of five scalars using a fixed format with values left-justified in their fields by the following:

```
SCALAR V,W,X,Y,Z
READ [LAYOUT=fixed;JUSTIFIED=left] V,W,X,Y,Z; \
     FIELDWIDTH=4,5,7,4,5
1.235.62␣678.9␣␣3.7810.31:
```

This reads the values 1.23, 5.62, 678.9, 3.78 and 10.31 into `V`, `W`, `X`, `Y` and `Z` respectively.

The general principles of the `SKIP` option and parameter are discussed in the context of a free format read in the previous section. When reading in fixed format the same ideas apply, but the `SKIP` settings now specify numbers of characters to be ignored, instead of numbers of values. Thus, you can obtain exactly the same effect as in the example above by putting

```
READ [LAYOUT=fixed] V,W,X,Y,Z; FIELDWIDTH=4,4,5,4,5; \
     SKIP=0,0,1,2,0
```

Sometimes fixed format data can be further compressed by omitting the decimal point. The `DECIMALS` parameter allows you to re-scale data automatically when it is read; details are given in 3.1.11.

When reading textual data in fixed format, the contents of each field are taken exactly as they appear in the input file. There is no need to enclose values in quotes; in fact if you do so, the quotes are treated as part of the data. For example,

```
TEXT [NVALUES=1] T1,T2,T3,T4
READ [LAYOUT=fixed; SKIP=*] T1,T2,T3,T4; FIELDWIDTH=6,3,4,7
'What's␣it␣all␣about?':
```

gives text `T1` the value `'What's`, text `T2` the value `␣it`, text `T3` the value `␣all`, and text `T4` the value `␣about?'`.

Consequently, the only way to represent a missing string in fixed format is by a blank field, as `''` or `*` would both be treated literally and stored as data values.

The `TRUNCATE` option allows you to remove unwanted spaces when reading texts or factors as labels. Setting `TRUNCATE=trailing` removes trailing blanks in each line of text. `TRUNCATE` also has a setting `leading` to delete initial blanks. `TRUNCATE` is particularly useful when reading labels of factors in fixed format. This is illustrated in Example 3.1.7, where the values of the factor `Country` are read as labels in a fixed field of seven characters. By setting `TRUNCATE=trailing` the extraneous spaces at the end of Canada and France are removed, allowing them to be recognized correctly.

Example 3.1.7

```
 2  TEXT   [VALUES=Austria,Belgium,Canada,Denmark,England,\
 3         France,Germany] Cname
 4  FACTOR [LABELS=Cname] Country
 5  READ   [LAYOUT=fixed; SKIP=*; TRUNCATE=leading,trailing] Country;\
 6         FIELDWIDTH=7; FREPRESENTATION=labels

  Identifier    Values   Missing    Levels
     Country         7         0         7

15  PRINT Country; JUSTIFICATION=right

   Country
   England
    Canada
    France
   Belgium
```

```
    Germany
    Austria
    Denmark
```

3.1.8 Reading data with variable formats

When you are responsible for producing your own data files you can ensure that they are arranged so that they can be read using simple combinations of the options and parameters of `READ`. Usually the default settings will be sufficient. However, when you obtain data from other sources this may not be the case. For example, you might find it necessary to read in fixed format as described in 3.1.7. Sometimes even this may not provide sufficient flexibility, so you can set the `FORMAT` option and use a *variable format*. By this we mean that the layout of the values may vary from unit to unit of the data, and may also vary within each unit. For example, suppose you have some meteorological data which was measured daily and that the file also contains some additional summary values at the end of each week. The first eleven lines are reproduced to illustrate the structure of the file:

```
    Monday          5.5      -0.4        0.0        1.9       10.0
    Tuesday        -1.1      -2.1        0.0        0.0       34.0
    Wednesday       0.6      -8.3        1.3        5.4      142.0
    Thursday        6.8      -5.7        1.1        0.0      158.0
    Friday         10.6       0.5        8.1        0.0      141.0
    Saturday       10.7       6.4        8.3        0.0      152.0
    Sunday         10.0       1.9        1.0        0.1      237.0
    Summary week 1>  10.7  -8.3  4  19.8  7.4  10.0  124.8  237.0
    Monday          9.9       2.5        0.0        4.4      229.0
    Tuesday        11.4       2.1        8.5        0.3      237.0
    Wednesday      11.9       6.3       18.7        0.0      520.0
```

Suppose the file contains data for 28 days. If you try to read a text and five variates of length 28 then the summaries found after the 7th, 14th, 21st and 28th days would cause an error in `READ`. You need to read seven lines, skip one, read seven more, and so on. This can be done by setting the option `FORMAT=!( (6)7,*,* )`. This means "read six values, do this seven times, skip to the next line, skip again, then return to the beginning of the format and repeat, until enough data has been read". The format is made clear by using `(6)7` which corresponds to the physical layout of the data, but `42` could have been specified instead, meaning read the next 42 values.

You can use `FORMAT` when reading in either free format or fixed format, and can also switch between the two during the `READ`. When you have set `FORMAT`, Genstat ignores the `SKIP` option and the `FIELDWIDTH` and `SKIP` parameters, and `READ` is controlled entirely by the values of the `FORMAT`. These values are not in parallel with the list of structures: they apply to data values in turn, recycling from the beginning when necessary.

You set `FORMAT` to a variate, which may be declared in advance or can be an unnamed structure as shown above. Each value of this variate is interpreted as follows (where *n* is a positive integer):

+*n*	read *n* values (in free format) or one value from a field of *n* characters (in fixed format);
-*n*	skip the next *n* values (in free format) or *n* characters (in fixed format)
*	skip to the beginning of the next line
0.0	switch to fixed format
0.1	switch to free format using space as a separator
0.2	switch to free format using comma as a separator
0.3	switch to free format using colon as a separator
0.4	switch to free format using semicolon as a separator
0.5	switch to free format using the setting of the `SEPARATOR` option

Using the `FORMAT` variate `READ` will start in either free format or fixed format, according to the setting of `LAYOUT` (by default, `LAYOUT=separated`; that is, free format). You can switch between these at any time by specifying a value in the range 0-0.5. Remember that if you use free format, spaces and tabs can also be used in addition to the specified separator, and you must use a separator that is distinct from the `END` and `MISSING` indicators (see 3.1.3).

3.1.9 Reading from a text structure

You can use `READ` to read data that has been stored in a text structure, by giving the identifier of the text as the setting of the `CHANNEL` option. Each string of the text is treated as a line of input, as if it had been read from a file. The length of each string defines the length of line that is read; this may vary from line to line, so you will find that reading in fixed format is rather difficult to specify correctly, and is perhaps better avoided here.

For example:

```
TEXT [VALUES=\
  '35 ''J. Smith'' 24000',\
  '24 ''G. Brown'' 11500:',\
  '22 33 44 55',\
  '66 55 77 88 :'] Data
TEXT Name
READ [CHANNEL=Data; SETNVALUES=yes] Age,Name,Income
& X
```

This gives `Age`, `Name` and `Income` each two values, and `X` eight.

Care is needed if you define the values of the text in the declaration as, in a string list, any sequences of the single-quote, double-quote or backslash characters will be halved in length when they are assigned to the text structure (1.4.2). In the example above, the first line that is stored in `Data` and then read is actually

```
35 'J. Smith' 24000
```

Just as when reading from a file, `READ` keeps a records of its current position when reading from a text, so that a subsequent `READ` from the same text will continue at the next line. This means that you can read more than one set of data from a text, but you too need to remember the position particularly when writing general programs or procedures. If you need to start again from the beginning you can set `REWIND=yes`, or you can use the `CLOSE` directive (3.3.2) to close the text. If the text is redefined, for example by a `TEXT`, `READ` or `CONCATENATE` statement, an implicit `CLOSE` is carried out, so that the input buffers are not inconsistent with the new values of the text.

3.1.10 Reading large data sets

You may sometimes have more data to read than can be stored in the space available within Genstat. You can then use the `SEQUENTIAL` option of `READ` to process the data in smaller batches. This works by reading in some of the data, partially processing it to form an intermediate result, and then overwriting the original data with a new batch that is used to update the intermediate results. This can be repeated until all the data has been read and the final summary is obtained. There are two directives that include facilities specifically designed to work with sequential data input: `TABULATE` which forms tabular summaries (4.11.1), and `FSSPM` which forms `SSPM` data structures for use in linear regression (4.10.3). You can also use other directives, such as `CALCULATE`, to process data sequentially, but you will have to program the sequential aspects yourself.

You should first declare the structures to be of some convenient size, such that you will not use up all the work space. You then use `READ` as normal, but with the `SEQUENTIAL` option set to the identifier of a scalar, which will be used to keep track of how the input is progressing. For example, to read in 10 variates of length 272500:

```
VARIATE [NVALUES=10000] X[1...10]
READ [CHANNEL=2; SEQUENTIAL=N] [1...10]
```

The number of values declared for `X[1...10]` defines the size of batch to read (10000 in this example). So, `READ` will read the first 10000 units of data (100,000 values), and set `N` to 10000 to indicate that is the number of units read. This should be followed by the statements to process the first batch of data, then the `READ` can be repeated. Once again `N` is set to 10000, indicating that another 10000 units have been read. This can be continued until `READ` finds the data terminator, when it sets the sequential indicator to minus the number of values found in the last batch. If this is less than the declared size of the data structures they will be filled out with missing values. In the example given above, after the 28th `READ` the variates will each contain 2500 values followed by 7500 missing values, and `N` will be set to `-2500`, indicating that all the data has been read and that the final batch contains only 2500 values. Usually you will use the `SEQUENTIAL` facility in conjunction with `FSSPM` or `TABULATE` which are designed to recognize the different settings of the scalar `N`.

The `SEQUENTIAL` option is best used within a `FOR` loop (5.2.1). You should set the `NTIMES` option to a value large enough to ensure that sufficient batches of data are read. The loop should contain the `READ` statement and any other statements required to process the data. For example

```
VARIATE [NVALUES=10000] X[1...10]
SSPM [TERMS=X[]] S
FOR [NTIMES=9999]
  READ [PRINT=*; CHANNEL=2; SEQUENTIAL=N] X[]
  FSSPM [SEQUENTIAL=N] S
  EXIT N.LE.0
ENDFOR
```

The `EXIT` directive is used to jump out of the loop once all the data has been read and processed; this is safer than trying to program an exact number of iterations for the loop. The exit condition includes the case when `N` is equal to zero, as this will arise when the batch size exactly divides the total number of units. In the above example, if there were 280000 units of data altogether, the 28th `READ` would terminate with `N` set to 10000. This is because `READ` is unable to look ahead for the terminator, as there may be other statements in the loop, such as `SKIP`, which affect how the file is read. The next `READ` would immediately find the data terminator, so would exit with `N` set to zero. This special case is treated appropriately by `FSSPM` and `TABULATE`, but you should remember to allow for it if you are programming the sequential processing explicitly.

You can use the `SEQUENTIAL` option to read data from more than one input channel, perhaps when a large data set is split into two or more files, but you are not allowed to read data from the current input channel (that is, the channel containing the `READ` statement). If you want to process several structures sequentially from the same file, you must read them in parallel. You must also be careful not to modify the value of the scalar, `N`, within the loop when using sequential data input with `FSSPM` or `TABULATE`, as that could interfere with the sequential processing.

Another means of handling large amounts of data is provided by the `ADD` option. This allows you to add values to those already stored in a structure, thus forming cumulative totals without having to store all the individual data values. You must set `SERIAL=yes` with `ADD=yes`; and it is allowed only for variates. For example:

```
VARIATE [NVALUES=6] A
READ [ADD=yes; SERIAL=yes] 3(A)
5 12 9 * * 9 :
8 1 3 * 2 10 :
3 4 0 * 11 * :
```

This starts by assigning the values 5, 12, 9, *, * and 9 to `A`. Then `A` is read again, and its values become 13, 13, 12, *, 2, 19: with `ADD=yes` (and only then) missing values are interpreted as zeroes when being added to non-missing values. Finally `A` contains the values 16, 17, 12, *, 13, 19.

When you read large quantities of data it may be worth using the `ERRORS` and `QUIT` options, described in 3.1.13, to control error recovery from `READ`.

3.1.11 Automatic re-scaling of data

You can scale values with the DECIMALS parameter. For example, suppose you put

```
READ [SETNVALUES=yes] A; DECIMALS=3
2523  2.1 376 0.78 :
```

The values of A would then be 2.523, 2.1, 0.376, 0.78. DECIMALS specifies a power of 10 by which any value that does not contain a decimal point is scaled down. Negative powers are not allowed.

3.1.12 Automatic sorting of data (using the UNITS structure)

If you have used the UNITS directive (2.3.4) to specify a variate or text containing unit labels, READ will respect the order of these values when reading other structures in parallel with the units structure; in other words the data is re-ordered to match the order of the unit labels. In Example 3.1.12 the unit structure Item is read in parallel with variate Stock. This does not alter the values of Item, but its values are used to indicate which unit of the data is being read, and thus the order in which to store the values of Stock.

Example 3.1.12

```
TEXT  [VALUES=Beans,Carrots,Peas,Sardines,Tuna] Cans
UNITS Cans
READ  [PRINT=data,errors] Cans,Stock
Tuna 2  Peas 3  Beans 4  Carrots 0  Sardines 6 :
PRINT Cans,Stock; DECIMALS=0

                Cans       Stock
     Cans
    Beans     Beans            4
  Carrots   Carrots            0
     Peas      Peas            3
 Sardines  Sardines            6
     Tuna      Tuna            2
```

If the units structure does not already have values, READ will define the order of the units as the order in which it finds them in the data. This means that if you are reading several sets of data, each having a column for the unit number (or label), the first use of READ will define the unit order and subsequent READ statements will ensure that this order is maintained consistently in the remaining data.

If a value is specified more than once when defining the units structure, READ will only ever locate the first occurrence of that unit label. If a unit label is repeated in the data then only the final set of values corresponding to that unit will be stored; earlier occurrences are overwritten by subsequent ones. If you try to read a value that is not present in the units structure this is regarded as a fault. Also, if the units structure contains missing values, it cannot be used to re-order the data and will instead be overwritten by the new values: a warning message is printed out to tell you if this occurs. If you use the option SETNVALUES=yes when reading structures in parallel with the units vector, the other structures will all be set to the current unit length.

3.1.13 Errors while reading

There are various kinds of error that may arise during execution of a READ statement. There are those that immediately inhibit the read, such as an attempt to read in a structure that is not sufficiently defined. For example, if you declare a matrix M, without specifying its dimensions, READ will not know how many values are required. Other examples include trying to read incompatible structures in parallel (for example variates of different lengths), or specifying a channel that has not been opened. If you make an error of this kind, READ will generate an appropriate diagnostic just like any other directive.

There are some checks that READ will make after it has read all the data. For example, it checks whether you have supplied the correct number of values, generating a fault if there are too many. If there are too few, the structures are completed with missing values and a warning is printed. If you are reading in parallel, this check is extended to ensure that the number of values supplied is a multiple of the number of structures. For example, suppose that values for five structures of length 10 are being read in parallel. If 45 values are found, then the structures will be completed with missing values; but if only 43 values are read in READ assumes that something more serious must be wrong with the data and generates a fault.

The rest of this section looks at errors that can arise while reading the data, and assumes that the READ statement has been specified correctly.

When you are working interactively and typing data at the terminal, READ will halt immediately it finds an invalid value. You should type the correct value and then continue with the rest of the data. If you had typed several items of data then all those before the erroneous value will have been read and stored, but any remaining values will have been discarded, and so will need to be retyped. For example, suppose you misspell a factor label:

Example 3.1.13a

```
> FACTOR [LABELS=!T(Avon,Bedford,Cornwall,Devon)] County
> READ County; FREPRESENTATION=labels
County/1> Avon Avon Cornwall
County/4> Bedford devon Cornwall :
******** Warning (Code IO 11). Unit 5 of County is incorrect.
Input:     devon      Code IO 44: Factor value not found in LABELS

Please input the correct value and subsequent data (the remainder of the last
line will be ignored).
County/5> Devon Cornwall :

    Identifier    Values   Missing    Levels
        County         6         0         4

>
```

The message indicates which unit is incorrect and also gives an explanation of the error (in this case devon was invalid because it should have started with a capital letter). The prompt indicates where READ is restarting its input; note that the value for the sixth unit has to be given again even though it was correctly specified in the original input.

When you are reading data in batch, it is not possible to recover from errors in this way. Instead, READ will continue processing the data, substituting missing values for any data that it cannot read, and printing out a message for every error that is found.

Example 3.1.13b

```
   2  VARIATE Speed
   3  FACTOR [LEVELS=!(30,40,50,70)] Limit
   4  READ Speed,Limit

******** Warning (Code IO 11). Statement 1 on Line 4

Command: READ Speed,Limit
Errors in data values.

 Unit Identifier   Input:
    1      Speed    l       Code SX 39: Invalid character in number.

    4      Limit    60      Code IO 43: Factor value not found in LEVELS.

    5      Speed    1.0e999
```

```
                          Code IO 3: Real number too large.

    Identifier   Minimum      Mean   Maximum    Values   Missing
         Speed     35.00     47.33     55.00         5         2

    Identifier    Values   Missing    Levels
         Limit         5         1         4

******** Fault (Code IO 8). Statement 1 on Line 4.

Command: READ Speed,Limit
Too many errors in data.

A fatal fault has occurred - the rest of this job will be ignored
```

The first value of `Speed` was incorrect as a letter `I` had been typed, rather than the number 1. Subsequent messages illustrate some of the other errors that may occur when reading data. Notice that the data summaries indicate the presence of missing values, which were inserted by `READ`. Of course, if you get errors when reading data it may be due to incorrectly specified options or parameters in the `READ` statement, rather than actual errors in the data file. This is especially likely if you are reading in fixed format or using the `FORMAT` option.

If errors occur when running in batch, a fault will be generated when `READ` terminates, thus terminating the job. This is to avoid spurious output being produced from analyses based on incorrect data. You can override this by using the options `ERRORS` and `QUIT`.

If you set `ERRORS=`*n*, where *n* is a positive integer, then up to *n* errors are allowed in the data before `READ` generates a fault. You might want to do this if you knew certain items of data were going to generate errors, but were prepared to accept them as missing values so that you could analyse the rest of the data. Obviously, you need to be very careful when doing this, as there may be other unexpected errors in the data. Usually you would have to try reading the data once without setting `ERRORS`, so you could check all the messages, and find what value of *n* is appropriate. Then the `READ` statement would have to be repeated, setting `ERRORS` and `REWIND` (3.1.3) in order to read the data. For example, if missing values of a factor had been typed in as the letter `X`, you would not want to define `X` as an extra level of the factor, but if you set `MISSING='X'` any numerical data that used `*` for missing value could not be read either.

As already explained, `READ` produces a message for every data value that contains an error. This can be very useful, as you then have the opportunity to correct all the errors at once, before trying to read the data again. However, the error messages may not be due to errors in the data, but may be caused by an incorrectly specified `READ` statement. For example, if you are reading many structures in parallel and specify texts and variates in the wrong order in the list of structures to be read, you will get an error message every time Genstat finds a piece of text rather than a number in the position specified for a variate. This is not likely to be a problem, unless you are reading large amounts of data, when you might end up with thousands of lines of needless error messages. A sensible precaution then is to request Genstat to abort the `READ` if more than a specified number of errors occur. You can do this by setting `ERRORS` to a negative integer, `-`*n*. This means that up to *n* errors are allowed in the data, but `READ` will abort if any more occur, switching control to the channel specified by `QUIT` (that is, starting or continuing to read Genstat statements from that channel). If you are working in batch a fault will be generated that inhibits execution of further statements, but interactively you have the opportunity to examine the data that have been read in so far, which may help identify any problems in the original `READ` statement or declarations of your data. For example:

Example 3.1.13c

```
> OPEN 'Data.dat'; CHANNEL=2; FILETYPE=input
> FACTOR [LABELS=!T(Die,Sand)] Casting
> VARIATE Breakage
> READ [CHANNEL=2; ERRORS=-3] Breakage,Casting

******** Warning (Code IO 11). Statement 1 on Line 4
Command: READ [CHANNEL=2;ERRORS=-3] Breakage,Casting
Errors in data values

 Unit Identifier   Input:
    1    Casting   Die       Code SX 39: Invalid character in number
    2    Casting   Die       Code SX 39: Invalid character in number
    3    Casting   Die       Code SX 39: Invalid character in number
    4    Casting   Sand      Code SX 39: Invalid character in number

******** Fault (Code IO 8). Statement 1 on Line 4
Command: READ [CHANNEL=2;ERRORS=-3] Breakage,Casting
Too many errors in data
3 allowed
> PRINT Breakage,Casting

    Breakage     Casting
       147.2           *
       119.1           *
       127.8           *
        97.3           *

> READ [CHANNEL=2; ERRORS=-3; REWIND=yes; SETNVALUES=yes]\
>      Breakage,Casting; FREPRESENTATION=labels

    Identifier   Minimum      Mean   Maximum    Values   Missing
      Breakage     61.20     131.1     164.6      1247         0
    Identifier    Values   Missing    Levels
       Casting      1247         0         2
```

The READ terminated after the fourth error in the data. Control returned to channel 1, the keyboard (using the default setting of QUIT). Printing out the two structures showed that they had been set up with four values, the number of units that had been completely read before quitting. All the errors had occurred in the factor values: in this case the mistake was easily identified, the FREPRESENTATION parameter had been omitted so that the default levels were expected rather than the labels which were in the data file. The READ statement was then repeated, specifying FREPRESENTATION=labels, and using REWIND to start again from the beginning of the file and SETNVALUES to reset their lengths.

3.2 Printing data

The contents of Genstat data structures can be displayed, with appropriate labelling, using the PRINT directive. In Genstat *for Windows*, PRINT is used by the Display Data in Output Window menu to display the values of data structures in the Output window. This menu is obtained by highlighting the structures of interest in the Data Display menu (obtained by pressing the F5 key), and then clicking the Display button. PRINT can also send output to other output channels, or put it into a text structure. PRINT has many options and parameters to allow you to control the style and format of the output but, in most cases, these can be left with their default settings.

These simple uses of PRINT are described in Section 3.2.1, while the more sophisticated features are in Section 3.2.2. Section 3.2.3 describes the CAPTION directive which prints titles in Genstat's standard formats, and Section 3.2.4 covers the PAGE directive which allows you to advance to a new page before starting the next section of output.

3.2.1 Main features of the PRINT directive

PRINT directive

Prints data in tabular format in an output file, unformatted file or text.

Options

`CHANNEL` = *identifier*	Channel number of file, or identifier of a text to store output; default current output file
`SERIAL` = *string token*	Whether structures are to be printed in serial order, i.e. all values of the first structure, then all of the second, and so on (`yes`, `no`); default `no`, i.e. values in parallel
`IPRINT` = *string tokens*	What identifier and/or text to print for the structure (`identifier`, `extra`, `associatedidentifier`), for a table `associatedidentifier` prints the identifier of the variate from which the table was formed (e.g. by `TABULATE`), `IPRINT=*` suppresses the identifier altogether; default `iden`
`RLPRINT` = *string tokens*	What row labels to print (`labels`, `integers`, `identifiers`), `RLPRINT=*` suppresses row labels altogether; default `labe`, `iden`
`CLPRINT` = *string tokens*	What column labels to print (`labels`, `integers`, `identifiers`), `CLPRINT=*` suppresses column labels altogether; default `labe`, `iden`
`RLWIDTH` = *scalar*	Field width for row labels; default 13
`INDENTATION` = *scalar*	Number of spaces to leave before the first character in the line; default 0
`WIDTH` = *scalar*	Last allowed position for characters in the line; default width of current output file
`SQUASH` = *string token*	Whether to omit blank lines in the layout of values (`yes`, `no`); default `no`
`MISSING` = *text*	What to print for missing value; default uses `'*'` for numbers and blanks in texts
`ORIENTATION` = *string token*	How to print vectors or pointers (`down`, `across`); default `down`, i.e. down the page
`ACROSS` = *scalar* or *factors*	Number of factors or list of factors to be printed across the page when printing tables; default for a table with two or more classifying factors prints the final factor in the classifying set and the notional factor indexing a parallel list of tables across the page, for a one-way table only the notional factor is printed across the page
`DOWN` = *scalar* or *factors*	Number of factors or list of factors to be printed down the page when printing tables; default is to print all other factors down the page
`WAFER` = *scalar* or *factors*	Number of factors or list of factors to classify the separate "wafers" (or slices) used to print the tables; default 0
`PUNKNOWN` = *string token*	When to print unknown cells of tables (`present`, `always`, `zero`, `missing`, `never`); default `pres`
`UNFORMATTED` = *string token*	Whether file is unformatted (`yes`, `no`); default `no`
`REWIND` = *string token*	Whether to rewind unformatted file before printing (`yes`, `no`); default `no`
`WRAP` = *string token*	Whether to wrap output that is too long for one line onto subsequent lines, rather than putting it into a subsequent "block" (`yes`, `no`); default `no`
`STYLE` = *string token*	Style to use for an output file (`plaintext`, `formatted`); default `*` uses the current style of the

	channel
PMARGIN = *string tokens*	Which margins to print for tables (full, columns, rows, wafers); default full
OMITMISSINGROWS = *string token*	Whether to omit rows of tables that contain only missing values (yes, no); default no
VSPECIAL = *scalar* or *variate*	Special values to be modified in the output
TSPECIAL = *text*	Strings to be used for the special values; must be set if VSPECIAL is set

Parameters

STRUCTURE = *identifiers*	Structures to be printed
FIELDWIDTH = *scalars*	Field width in which to print the values of each structure (a negative value -*n* prints numbers in E-format in width *n*); if omitted, a default is determined (for numbers, this is usually 12; for text, the width is one more character than the longest line)
DECIMALS = *structures*	Number of decimal places for numerical data structures, a scalar if the same number of decimals is to be used for all values of the structure, or a data structure of the same type and size to use different numbers of decimals for each value; if omitted or set to a missing value, a default is determined which prints the mean absolute value to 4 significant figures
CHARACTERS = *scalars*	Number of characters to print in strings
SKIP = *scalars* or *variates*	Number of spaces to leave before each value of a structure (* means a new line before structure)
FREPRESENTATION = *string tokens*	How to represent factor values (labels, levels, ordinals); default is to use labels if available, otherwise levels
JUSTIFICATION = *string tokens*	How to position values within the field (right, left, center, centre); if omitted, right is assumed
MNAME = *string tokens*	Name to print for table margins (margin, total, nobservd, mean, minimum, maximum, variance, count, median, quantile); if omitted, "Margin" is printed
DREPRESENTATION = *scalars* or *texts*	Format to use for dates and times (stored in numerical structures)
HEADING = *texts*	Heading to be used for vectors printed in columns down the page; default is to use the information requested by the IPRINT option
TLABELS = *texts*	If this is specified for a table STRUCTURE, the values of the table are interpreted as references to lines within the TLABELS text that are to be printed instead of the values of the table itself

For a quick display of the contents of a list of data structures, you need only give the name of the directive, PRINT, and then list their identifiers. For example,

```
    PRINT Source,Amount,Gain
```

The output is fully annotated with the identifiers, and with row and column labels or numbers, where appropriate. Factors are represented by their labels if available, and otherwise by their levels. The layout of the values is determined automatically by the size and shape of the structures to be printed, and by the space needed to print individual values. The output is arranged in columns; the structures are split if the page is not wide enough, so that one set of columns is completed before the next is printed. Example 3.2.1a prints the values of two factors, `Source` and `Amount`, and a variate `Gain`.

Example 3.2.1a

```
   2  UNITS [NVALUES=12]
   3  FACTOR [LABELS=!T(beef,cereal,pork); \
   4    VALUES=1,3,2,3,1,2,2,1,3,1,2,3] Source
   5  & [LEVELS=!(25,50); LABELS=!T(low,high); \
   6    VALUES=50,25,50,50,25,25,50,25,50,50,25,25] Amount
   7  VARIATE [VALUES=73,49,98,94,90,107,74,76,79,102,95,82] Gain
   8  PRINT Source,Amount,Gain

      Source      Amount        Gain
        beef        high       73.00
        pork         low       49.00
      cereal        high       98.00
        pork        high       94.00
        beef         low       90.00
      cereal         low      107.00
      cereal        high       74.00
        beef         low       76.00
        pork        high       79.00
        beef        high      102.00
      cereal         low       95.00
        pork         low       82.00
```

As the three vectors all contain the same number of values, the default is to print their values in parallel. Alternatively, you can request that structures are printed in series, one below another, by setting option `SERIAL=yes`. Of course, if the structures to be printed have different shapes or sizes, their values can be printed only in series. The setting `SERIAL=no` is then ignored except that, to save space, any vectors or pointers are then printed across the page (that is as though you had set `ORIENTATION=across`: see Example 3.2.1e). Genstat annotates each set of values by the identifier of the structure (but this can be controlled by the `HEADING` parameter or the `IPRINT` option, described below) and automatically chooses a suitable format.

You can use the `RESTRICT` directive (4.4.1) to specify that only a subset of the units of a vector should be printed. When printing in series the vectors can be restricted to different subsets; but with parallel printing any restriction is applied to all the vectors (and any pointers) so, if more than one vector is restricted, they must all have been restricted in the same way.

Genstat can produce output in either plain-text or a "formatted" style written in either RTF, HTML or LaTeX. The style is set when the channel is opened, either by the `OPEN` directive (3.3.1) or by the command used to run Genstat (1.1.2). You can also switch a formatted output channel temporarily into the plain-text style (and back into its formatted style) using the `OUTPUT` directive (3.4.3).

Plain-text output assumes that all characters occupy an equal width, so columns are aligned using space characters. The other styles use special codes to define the columns. However, you can set option `STYLE=plain` to request that output to files with these other styles should use spaces instead (i.e. `PRINT` then operates as though they were in plain-text style). This is useful particularly in procedures (5.3.2), when you may want to print a "sentence" containing information from several different data structures.

In plain-text output, the default for a numerical structure is to use a field of *f* characters. Generally, the value of *f* is 12, but another value can be defined using the `FIELDWIDTH` option of the `SET` directive (5.6.1). Labels of factors are usually printed in a field of 12 characters but this is extended if any of the strings in the text requires a wider field. Texts are printed in a field one larger than the width of their longest line. With formatted output, the field width defines the fraction of the full line width to be used. So, for example if the line width (defined by the `WIDTH` option of `PRINT`) is 80, a field width of 10 indicates that the structure should use 1/8th of the line.

If the `DECIMALS` parameter was set when a numerical structure was declared (2.1.2), this will define the number of decimal places in the output. Otherwise, the number of decimal places is usually determined by calculating the number that would be required to print its mean absolute value to at least *d* significant figures. Generally, *d* is four, but this can be redefined using the `SIGNIFICANTFIGURES` option of the `SET` directive (5.6.1).

Alternatively, you can define your own formats using the parameters `FIELDWIDTH`, `DECIMALS`, `CHARACTERS`, `SKIP` and `JUSTIFICATION`. The `DECIMALS` and `MINFIELDWIDTH` procedures may then be helpful; see 3.2.5 and 3.2.6 for details..

Example 3.2.1b

```
  9  PRINT Source,Amount,Gain; FIELDWIDTH=7,7,6; DECIMALS=*,*,0

Source Amount  Gain
  beef   high    73
  pork    low    49
cereal   high    98
  pork   high    94
  beef    low    90
cereal    low   107
cereal   high    74
  beef    low    76
  pork   high    79
  beef   high   102
cereal    low    95
  pork    low    82
```

Example 3.2.1b illustrates the use of the parameters `FIELDWIDTH` and `DECIMALS`. `FIELDWIDTH` indicates the field width to use to print each data structure; a negative value, of $-f$ say, prints numbers in scientific format (for example `7.3 E1` for the first unit of `Gain`) in a width of *f* with `DECIMALS` significant places. You can set `DECIMALS` to a scalar to use the same number of decimals for all the values of a numerical data structure. Alternatively, if you want to use a different number of decimals for each value, you can supply a data structure of the same type and size as the data structure; see Example 3.2.1c. If `DECIMALS` contains a missing value, a default is used which prints the mean absolute value to *d* significant figures, as explained above. The `DECIMALS` parameter is ignored for strings, like the labels of the factors `Source` and `Amount`. (So in line 9, we could just have put `DECIMALS=0`, instead of `DECIMALS=*,*,0`.)

In the same way, the `CHARACTERS` parameter is ignored for numbers; for strings, it allows you to control the number of characters that are printed. So, we could put `CHARACTERS=1` in Example 3.2.1b to print only the first letter of each factor label. By default, Genstat prints all the characters in each string of a text or factor label, unless the `CHARACTERS` parameter was set to a lesser number when the text or factor was declared (2.3.3).

The `SKIP` parameter allows you to place extra spaces between the values of each structure. By default, no extra spaces are inserted unless a value fills the field completely, when a single space will be inserted; there is also a blank line before the first printed line. `SKIP` can be set to either a scalar or a variate in which a positive integer *n* requests that *n* spaces are left and a missing value can be used to request a blank line. So, for example, we could put `SKIP=0,2,2` to move the columns in Example 3.2.1b two further spaces apart. The zero value for `Source` would mean that there were no extra spaces to the left of the block of output. There would also be no blank line before the output. This can be reinstated by specifying a scalar (or variate) containing a missing value in the `SKIP` setting for `Source`. However, there is the limitation that these missing values are ignored for the second and subsequent structures when printing in parallel.

Example 3.2.1c

```
  10  PRINT Source,Amount,Gain; FIELDWIDTH=7,7,6; DECIMALS=Decimals;\
  11    FREPRESENTATION=labels,levels; \
  12    JUSTIFICATION=left,centre,right  &  '(measurements in grams)'

Source  Amount  Gain
beef    50.00  73.00
pork    25.00  49.00
cereal  50.00  98.00
pork    50.00  94.00
beef    25.00  90.00
cereal  25.00  107.0
cereal  50.00  74.00
beef    25.00  76.00
pork    50.00  79.00
beef    50.00  102.0
cereal  25.00  95.00
pork    25.00  82.00


 (measurements in grams)
```

The values can be left-justified by setting the `JUSTIFICATION` parameter to `left` as has been done for the factor `Source` in Example 3.2.1c, or centred by setting it to either `center` or `centre` as has been done for the factor `Amount`. This example also shows how to use the `FREPRESENTATION` parameter to control the printing of the factor values. By default Genstat will print labels if there are any; if there are none, it prints the levels. In the example, labels are printed for `Source`, levels are printed for `Amount`, and `FREPRESENTATION` is ignored for the variate `Gain`. The other available setting, `ordinals`, would represent the values by the integers 1 upwards; so for example `beef`, `cereal` and `pork`, would be represented by the numbers 1, 2 and 3, respectively. Line 12 shows how you can insert a caption into your output, by printing a string.

The default setting, `IPRINT=identifier`, will usually label the output with the identifier of the structure. However, this default can be modified by setting the `IPRINT` option when the data structure is declared; see Section 2.1.3. Putting `IPRINT=identifier,extra` will also include any "extra" text that has been associated with the structure by the `EXTRA` parameter when it was declared, while putting `IPRINT=extra` will use only the extra" text. The setting `associatedidentifier` can be used when a table has been produced by the `TABULATE` (4.11.1) and `AKEEP` (2:4.6.1) directives, to request that the output be labelled with the identifier of the variate from which the table was formed.

The `HEADING` parameter is useful when you want to use something other than the identifier of a variate, factor or text to label its column. In Example 3.2.1d, the string `'Source of protein'` is used to label the column for `Source`, and `'Weight gain'` to label the column for `Gain`. No heading is supplied for `Amount`, so this is labelled by its identifier. The heading, if supplied, simply replaces the identifier, and its appearance in the output is controlled by the `identifier` setting of `IPRINT` just like the identifier itself.

Example 3.2.1d

```
  13  PRINT Source,Amount,Gain; DECIMALS=0; SKIP=0,2,2;\
  14    HEADING='Source of protein',*,'Weight gain'
```

```
Source of protein        Amount   Weight gain
             beef          high            73
             pork           low            49
           cereal          high            98
             pork          high            94
             beef           low            90
           cereal           low           107
           cereal          high            74
             beef           low            76
             pork          high            79
             beef          high           102
           cereal           low            95
             pork           low            82
```

Example 3.2.1e illustrates the `ORIENTATION` option, which is relevant only when you are printing vectors or pointers. By setting `ORIENTATION=across`, the values are printed in alternate lines, across the page. To ensure that these line up correctly, the fieldwidth is taken as the maximum of those specified for the printed structures, while the field used to print their identifiers is given by the `RLWIDTH` option (by default 13).

Example 3.2.1e

```
PRINT [ORIENTATION=across; RLWIDTH=8] Source,Amount,Gain;\
  FIELDWIDTH=7,7,6; DECIMALS=0

Source   beef    pork cereal   pork   beef cereal cereal   beef   pork   beef
Amount   high     low   high   high    low    low   high    low   high   high
  Gain     73      49     98     94     90    107     74     76     79    102

Source cereal    pork
Amount    low     low
  Gain     95      82
```

Notice that Genstat now has to print the output in more than one block. This will happen whenever there is too much output to fit across the page, unless option `WRAP` is set to `yes`. Then Genstat simply wraps each line onto subsequent lines. This is likely to be useful mainly if you are printing the contents of the structures to be read by another program. You might then also wish to suppress the identifiers by setting option `IPRINT=*` and remove blank lines by setting option `SQUASH=yes`.

The width of each line can be controlled by the `WIDTH` option; the default is to take the full available width. The `INDENTATION` option specifies the number of spaces to leave before each line; by default there are none.

There are two other options that apply to any type of structure. The `CHANNEL` option determines where the output appears. By default, the output is placed in the current output channel, but `CHANNEL` can be set to a scalar to send it to another output channel; the correspondence between channels and files on the computer is described in 3.3. Alternatively, you can set `CHANNEL` to the identifier of a text to store the output. The text need not be declared in advance; any undeclared structure that is specified by `CHANNEL` will be defined automatically as a text. Each line of output becomes one value of the text and if the text already has values they will be replaced. You are most likely to want to do this in order to manipulate the text further. Remember, however, that if you print the text later on, its strings will be right-justified by default, so you will need to set `JUSTIFICATION=left` in the later `PRINT` statement to achieve the normal appearance of your output. The maximum (and default) line length of this text is the length of what is called the *output buffer*. This is likely to be 200 on most computers. If you intend to print it to an output file, you should set the `WIDTH` option as appropriate.

The `MISSING` option allows you to specify a string to represent missing values, instead of the default that uses the asterisk symbol for missing numbers, and blanks for missing values in texts.

For example, you could set `MISSING='unknown'` or `MISSING=' '`.

The `VSPECIAL` and `TSPECIAL` options allow you to substitute textual strings for other values of numerical structures. The values are specified, in either a scalar or a variate, using the `VSPECIAL` option. The `TSPECIAL` option then specifies a text, with as many values as the `VSPECIAL` scalar or variate, to define the strings to be printed instead. For example, in the following program, values of `prob` less than 0.001 are set to −1, and then printed as `'<0.001'`.

```
CALCULATE prob = prob * (prob.GE.0.001) - (prob.LT.0.001)
PRINT [VSPECIAL=-1; TSPECIAL='<0.001'] prob
```

`PRINT` can similarly be used for the straightforward printing of tables and the various types of matrix, as well as formulae and expressions. The options and parameters that control the layout of multi-way structures are described in 3.2.2, while 3.7 explains the `UNFORMATTED` and `REWIND` options which are used to send output to unformatted files.

3.2.2 Printing of multi-way structures

`PRINT` can easily be used to print matrices and tables, by taking the default layout and labelling. Examples of a two-way table and of a three-way table are shown in 2.5. For tables with more than one dimension, the usual layout has one factor across the page and the others down the page (see Example 2.5a); tables with only one dimension are printed down the page. Several tables can be printed in parallel, provided they all have the same classifying factors. As shown in Example 3.2.2a, the tables are then printed in alternate columns, as though they formed a larger table with an extra factor (called the table-factor) representing the list of tables. This extra factor thus becomes another (in fact, the final) factor to be printed across the page.

Example 3.2.2a

```
FACTOR [LEVELS=2] Lab
& [LEVELS=3; LABELS=!T(beef,cereal,pork)] Source
& [LEVELS=!(25,50); LABELS=!T(low,high)] Amount
TABLE [CLASSIFICATION=Lab,Source,Amount; \
   VALUES=162.4,171.2,173.6,160.8,148.4,157.6, \
          154.4,168.8,149.8,159.0,170.4,160.4] Startwt
& [VALUES=243.6,286.8,260.4,286.2,222.6,281.4, \
          231.6,313.2,217.2,255.0,249.6,315.6] Finalwt
PRINT Startwt,Finalwt

            Amount          low                     high
                        Startwt     Finalwt     Startwt     Finalwt
   Lab      Source
     1        beef        162.4       243.6       171.2       286.8
            cereal        173.6       260.4       160.8       286.2
              pork        148.4       222.6       157.6       281.4
     2        beef        154.4       231.6       168.8       313.2
            cereal        149.8       217.2       159.0       255.0
              pork        170.4       249.6       160.4       315.6
```

This default layout can be changed using the `ACROSS`, `DOWN` and `WAFER` options. You may wish to do this simply by changing the factors which appear down and across the page. The `ACROSS` option can be set to a scalar to specify how many factors should be printed across the page, or to a list of factors to say which ones they should be. `DOWN` similarly specifies the factors to be printed down the page. However, you cannot specify a list of factors for one of these options and a scalar for any of the others. The table-factor can be represented in these lists by inserting a `*` in the required position; if you do not mention the table-factor in either list it remains as the last factor in the `ACROSS` list. In Example 3.2.2b the table-factor, `Lab`, and `Amount` are printed across the page (in that order), and `Source` is printed down the page.

Example 3.2.2b

```
  11  PRINT [ACROSS=*,Lab,Amount; DOWN=Source] Startwt,Finalwt;F=8

         Startwt                         Finalwt
     Lab       1               2               1               2
  Amount     low    high     low    high     low    high     low    high
  Source
    beef   162.4   171.2   154.4   168.8   243.6   286.8   231.6   313.2
  cereal   173.6   160.8   149.8   159.0   260.4   286.2   217.2   255.0
    pork   148.4   157.6   170.4   160.4   222.6   281.4   249.6   315.6
```

The WAFER option allows you to split the output up into subtables or "wafers". This is particularly useful if the tables have many classifying factors, or if the factors have very long labels. The setting can again be either a scalar or a list of factors (possibly including the table-factor). As shown in Example 3.2.2c, each subtable has a heading its position in the full table. If the table-factor is included in the wafer, the identifier of the appropriate table will be printed at the beginning of the label for that wafer; this does not mean that the table-factor itself has been moved, simply that the labelling has been rearranged to make it easier to read.

Example 3.2.2c

```
  12  PRINT [ACROSS=Amount; DOWN=Lab; WAFER=Source] Startwt,Finalwt

Source beef.

       Amount         low                     high
                  Startwt     Finalwt     Startwt     Finalwt
          Lab
            1       162.4       243.6       171.2       286.8
            2       154.4       231.6       168.8       313.2


Source cereal.

       Amount         low                     high
                  Startwt     Finalwt     Startwt     Finalwt
          Lab
            1       173.6       260.4       160.8       286.2
            2       149.8       217.2       159.0       255.0


Source pork.

       Amount         low                     high
                  Startwt     Finalwt     Startwt     Finalwt
          Lab
            1       148.4       222.6       157.6       281.4
            2       170.4       249.6       160.4       315.6
```

You need not specify all the options DOWN, ACROSS and WAFER. If you leave any of them out PRINT will deduce the missing information.

When a table has margins, usually they will all be printed. However, you can control which are printed, by specifying the following settings of the PMARGIN option:

full	print all margins (default),
columns	print margins over column factors,
rows	print margins over row factors, and
wafers	print margins over wafer factors.

The OMITMISSINGROWS option also operates only on tables; if you set it to yes, PRINT will omit any lines of output where the tables contain only missing values.

You can control the space allowed for labels of the `DOWN` factors by using the `RLWIDTH` option. By default this is set to 13, but you might want something else if the labels are very small. If the width provided (by you, or implicitly) is inadequate, `PRINT` automatically resets it to accommodate the longest row label. The labelling of rows by the down factors is controlled by the `RLPRINT` option. The default, `RLPRINT=labels,identifiers`, prints the identifiers of the factors and their levels or labels. Similarly, the `CLPRINT` controls the labelling of columns by the across factors.

When tables are produced by `TABULATE` (4.11.1) Genstat sets an internal indicator for use by `PRINT` to indicate the appropriate label for any margins. When a single table is printed this name will be used by default. When printing tables in parallel, if they all have the same setting of the margin name indicator, the appropriate name is used. If they have different settings, or none at all (tables from sources other than `TABULATE`) the margins will be labelled `Margin` by default. You can change the label by setting the `MNAME` parameter. Tables printed in parallel must have the same label throughout, and Genstat will take the one specified for the first table in the list. But in serial printing, you can use a different margin name for each table.

The `TABULATE` (4.11.1) and `AKEEP` (2:4.6.1) directives also record the identifier of the variate from which the table was formed, and you can request that this be used to label the output, instead of the identifier of the table itself, by setting the `IPRINT` option to `associatedidentifier`.

The `PUNKNOWN` option controls the printing of the "unknown" cell of a table (see 3.5). The default action is to print this cell, labelled with the table identifier, but only if it contains a value other than missing value or zero. You can select one of five settings:

`present` (default)	print value if not missing or zero
`always`	print the unknown cell regardless of value
`zero`	print unless the value is zero
`missing`	print unless the value is missing
`never`	do not print the unknown cell whatever its value

Genstat tables can only contain numbers. However, you can use the `TLABELS` parameter to print tables of textual strings. You first need to form a Genstat text structure containing all the strings that may occur. Then form a table with the required classifying factors and, in each cell of the table, put the number of the line (within the text) of the string that you want to print there. This is illustrated Example 3.2.2d, where the numbers 4, 2 and 4 in the table `MainDirection` refer to the 4th, 2nd and 4th elements of the text `Direction`.

Example 3.2.2d

```
FACTOR [LABELS=!t(April,May,June)] Month
TEXT   [VALUES=North,South,East,West] Direction
TABLE  [CLASSIFICATION=Month; VALUES=12.2,5.8,10.7] MeanSpeed
&      [VALUES=4,2,4] MainDirection
PRINT  MeanSpeed,MainDirection; TLABELS=*,Direction

             MeanSpeed MainDirection
     Month
     April      12.200        West
       May       5.800       South
      June      10.700        West
```

Options `ACROSS`, `DOWN`, `WAFER`, `RLPRINT` and `CLPRINT` also apply to matrices. By default, though, if you have several matrices they will be printed one after another on the page.

With symmetric matrices the only options of these that are relevant are `RLPRINT` and `CLPRINT`; a further setting `integer` is available for these to request that the rows or columns be labelled by the integers 1 onwards, as well as, or instead of the labels provided with the symmetric matrix: for example setting `RLPRINT=integers` and `CLPRINT=integers, labels` would identify the rows by integers and the columns with integers and labels.

3.2.3 The `CAPTION` directive

`CAPTION` directive

Prints captions in standardized formats.

Option

`PFIRST` = *string tokens*	What to print first (`dots`, `page`, `outprint`); default `*` i.e. none

Parameters

`TEXT` = *texts*	Contents of the captions
`STYLE` = *string tokens*	Style for each caption (`plaintext`, `stress`, `minor`, `major`, `meta`, `note`, `status`); default `plai`

The `CAPTION` directive allows captions to be printed in the standard Genstat styles. The contents of the caption are supplied by the `TEXT` parameter. The `STYLE` parameter specifies a string to indicate the caption style:

`plaintext`	ordinary text,
`stress`	text to be emphasized,
`minor`	a minor caption signifying a sub-section in the output,
`major`	a major caption signifying a section in the output,
`meta`	a meta-caption to group several sections of output,
`note`	a "note" to the user, and
`status`	a "status" message.

The `PFIRST` option allows you to start the caption on a new page or to precede it by a line of dots (or a horizontal "rule" if the output is formatted; see the `OPEN` directive, 3.3.1). Alternatively, the `outprint` setting generates the dots or new page according to the setting for the current output channel (see the `OUTPUT` directive, 3.4.3).

The major, minor and plain-text captions are illustrated in Example 3.2.3.

Example 3.2.3

```
   2  TEXT    [VALUES=\
   3          'Notice that, in plain text captions, Genstat reformats the',\
   4          'lines of the text to fill each line of output and start',\
   5          'the next line at the end of a word.'] Text
   6  CAPTION [PFIRST=dots] 'Major heading','Minor heading',Text;\
   7          STYLE=major,minor,plaintext

7.........................................................................

Major heading
=============

Minor heading
-------------

Notice that, in plain text captions, Genstat reformats the lines of the text to
fill each line of output and start the next line at the end of a word.
```

3.2.4 The PAGE directive

PAGE directive

Moves to the top of the next page of an output file.

Option

CHANNEL = *scalar*	Channel number of file; default * i.e. current output file

No parameters

When output is to a file, graphs and output from statistical analyses will automatically start on a new page, unless you have requested otherwise using the OUTPRINT option of JOB (5.1.1) or SET (5.6.1). With other directives, such as PRINT or TABULATE, you can request a new page using the PAGE directive. By default, PAGE works on the current output channel, but you can use the CHANNEL option if you are sending output to another file.

PAGE has no effect unless output is to a file, and it achieves its effect by printing a line consisting of just the control code for a form feed (ASCII character 12). The effect of PAGE is therefore independent of the page size set by the OPEN directive (3.3.1).

3.2.5 The DECIMALS procedure

DECIMALS procedure

Sets the number of decimals for a structure, using its round-off (A. Keen).

Options

SETATTRIBUTE = *string token*	Attributes to be redefined for STRUCTURE (decimals); default deci
SIGNIFICANTFIGURES = *scalar*	Required number of significant figures; default takes the system default, which can be modified by SET

Parameters

STRUCTURE = *identifiers*	Numerical structure for which the number of decimals is to be set
DECIMALS = *scalars*	To save the number of decimals to use for all the values of each structure
ROUND = *scalars*	To save the round-off provided by using DECIMALS decimal places
VDECIMALS = *structures*	To save numbers of decimals for every value of each structure
VROUND = *structures*	To save the round-off for every value of each structure

As explained in 3.2.1, the number of decimals that Genstat uses as a default, when printing a numerical structure, is calculated as the number required to display the mean of the absolute values of the numbers in the structure to a standard number of significant figures. Usually the standard number of significant figures is four, but this "system default" can be changed using the SIGNIFICANTFIGURES option of the SET directive. The default method allows output to be generated automatically with reasonable accuracy. However, it may be preferable to use fewer decimals if the numbers can be represented exactly with three or fewer significant figures. For example it may be preferable to use two decimal places rather than four for a variate containing the values 0.1 and 0.21 (i.e. to print 0.10 and 0.21, rather than 0.1000 and 0.2100).

The DECIMALS procedure operates similarly to the standard Genstat default, except that the number of decimal places is decreased if the final decimal position would contain the digit zero for every value of the structure. It also differs in that it has its own SIGNIFICANTFIGURES option to specify the required number of significant figures from the system default.

The numerical structure for which the number of decimals is to be determined must be supplied using the STRUCTURE parameter. The DECIMALS parameter can save the appropriate number of decimal places (as a scalar), and parameter ROUND can save the maximum round-off over the values of the structure. By default DECIMALS modifies the declaration of the STRUCTURE so that this becomes its default number of decimal places for subsequent printing (see the DECIMALS parameter of SCALAR, VARIATE, TABLE, MATRIX and SYMMETRICMATRIX). However, you can set option SETTATTRIBUTE=* if you want the default number of decimals to remain unchanged.

DECIMALS can also calculate a separate number of decimal places for each of the values of the STRUCTURE. This can be saved (in a structure of the same type as the STRUCTURE) using the VDECIMALS parameter, and the round-off for each value can similarly be saved using the VROUNDOFF parameter.

The difference between the DECIMALS and VDECIMALS parameters is illustrated in Example 3.2.5.

Example 3.2.5

```
  VARIATE  [VALUES=1,0.1,0.01,0.001] X
  DECIMALS X; DECIMALS=dpt; VDECIMALS=vdpt
  PRINT    X,X; DECIMALS=dpt,vdpt

         X           X
     1.000           1
     0.100         0.1
     0.010        0.01
     0.001       0.001
```

3.2.6 The **MINFIELDWIDTH** procedure

MINFIELDWIDTH procedure

Calculates minimum field widths for printing data structures (R.W. Payne).

Option

IPRINT = *string tokens*	What identifier and/or text to print for the structure (identifier, extra); default is to take the IPRINT setting of each STRUCTURE

Parameters

STRUCTURE = *identifiers*	Data structures to be printed
FIELDWIDTH = *scalars*	Saves the minimum field widths
DECIMALS = *scalars*	Number of decimal places to be used for numerical data structures; if unset, a default is obtained using the DECIMALS procedure
SKIP = *scalars*	Number of spaces to leave before each value of the structure; default 1
FREPRESENTATION = *string tokens*	How to represent factor values (labels, levels, ordinals); default is to use labels if available, otherwise levels

MINFIELDWIDTH can be used to calculate the minimum field width that would be required to print a data structure in an even column down the page using the PRINT directive. The data structures are specified by the STRUCTURE parameter, and can be any of those supported by Genstat. The calculated field width is saved, in a scalar, by the FIELDWIDTH parameter.

The IPRINT option indicates how the values of each STRUCTURE are to be labelled. The identifier setting uses the identifier of the STRUCTURE, while the extra setting used the information that can be specified by the EXTRA parameter when data structures are defined by directives like VARIATE, FACTOR and TEXT. You can set IPRINT=* to indicate that the values are not to be labelled by either of these. Alternatively, if IPRINT is not specified, the default is taken from the IPRINT attribute of the STRUCTURE (which can be set by the IPRINT option of VARIATE, FACTOR, TEXT etc). This is the same default that is used by PRINT if its own IPRINT option is not specified.

With numerical structures, like variates or matrices, the DECIMALS parameter specifies the number of decimal places that are to be used. If you set DECIMALS to a scalar containing a missing value, the DECIMALS procedure is used by MINFIELDWIDTH to determine a default number of decimal places, and this is stored in the scalar so that you can use it later. The DECIMALS procedure is also used to obtain a default if the DECIMALS parameter is not set.

The SKIP parameter specifies how many spaces are to be left before each element of each STRUCTURE; default 1.

The FREPRESENTATION parameter controls the printing of the factor values. The default is to print labels if there are any; if there are none, it is assumed that levels will be printed. The ordinals setting represents the values by the integers 1 upwards.

Example 3.2.6 uses MINFIELDWIDTH to set default formats for the factors Source and Amount, and the variate Gain, from Example 3.2.1a.

Example 3.2.6

```
  18  SCALAR Dec
  19  MINFIELDWIDTH Source,Amount,Gain; FIELDWIDTH=Fs,Fa,Fg; DECIMALS=*,*,Dec
  20  PRINT Fs,Fa,Fg,Dec

          Fs          Fa          Fg          Dec
       7.000       7.000       5.000            0

  21  PRINT Source,Amount,Gain; FIELDWIDTH=Fs,Fa,Fg; DECIMALS=*,*,Dec

 Source Amount Gain
   beef   high   73
   pork    low   49
 cereal   high   98
   pork   high   94
   beef    low   90
 cereal    low  107
 cereal   high   74
   beef    low   76
   pork   high   79
   beef   high  102
 cereal    low   95
   pork    low   82
```

3.3 Accessing external files

Genstat makes use of various types of file. These are classified according to the information that they store. Some files are in the standard text format recognized by many other programs such as editors, which you can use to prepare your Genstat program and data files. Other files (binary files) are produced by Genstat in formats specific to Genstat. Graphics output files use standard formats which may use either ordinary text files or unformatted binary files.

Genstat accesses the files via *channels*. For each type there is a set of numbered channels that can be used to reference different files in the relevant directives. For example, there are five input channels, numbered 1 up to 5. Likewise, there are five output channels. Genstat distinguishes between the different types of channel, so you can have one file attached to output channel 3 and a different file simultaneously attached to backing store channel 3. Then, setting the option `CHANNEL=3` in `PRINT` and `STORE` statements will send the different kinds of output to the appropriate files. The table below gives details of the channel numbers that are generally available in most versions of Genstat. It is possible that a particular version may allow additional channels to be used for some types of file; if so, details should be in your local documentation. Graphics channels use a slightly different numbering system, in which the channel corresponds to the number of the graphics device (set by the `DEVICE` directive; 6.9.1).

Type of file	**Channels**	**Purpose**
input	1...5	text files containing Genstat instructions
output	1...5	text files to contain Genstat output
backingstore	0...5	structured binary files for storage of data
procedure library	1...3	backing store files containing procedures to be accessed automatically (5.3.3)
graphics	device numbers	text or binary files for storing graphical output that can subsequently be printed on a plotter or laser printer (6.9.1)
unformatted	0...5	binary files for rapid storage and retrieval of data

When you run Genstat it starts taking input from input channel 1 and produces output on output channel 1. In an interactive run, these will be keyboard and screen, while in a batch run they will be files on the computer (1.1). Another file that is attached automatically is the start-up file of instructions that are executed at the outset of each job (5.6.4); this is attached to input channel 5. The start-up file may attach other files, for example to hold a transcript of input or output; your local documentation will contain details.

The command that you use to run Genstat may allow you to arrange for other files to be attached when Genstat starts running. Alternatively, within Genstat, you can use the `OPEN` directive. `OPEN` also lets you define additional characteristics of the file, such as the maximum length of each line or the style for output. When you have finished using a file you can tell Genstat to `CLOSE` it (but note that all files are automatically closed by the `STOP` directive).

The `SKIP` directive can be used to skip over part of an input file or to print extra blank lines in an output file (3.3.3). The `ENQUIRE` directive (3.3.4) can be used to find out about the files that are currently connected to any of Genstat's channels; this is likely to be useful particularly within general programs and procedures.

One special type of file is the Genstat text structure, which may be thought of as an "internal" file. This can be used for input or output by certain directives. Those directives that can create texts to contain their output (for example `TEXT` or `PRINT`) leave them in a "closed" state. There is no need to open a text explicitly if you want to use it for input (for example in `READ`), but you may need to `CLOSE` it afterwards; see 3.1.9 for further details. The contents of the texts are lost at the end of the job unless you save them, for example by using backing store (3.5). The use of texts as internal files is a more advanced facility that is likely to be required only for more complicated programs and procedures.

3.3.1 The `OPEN` directive

`OPEN` directive
Opens files.

No options

Parameters

`NAME` = *texts*	External names of the files
`CHANNEL` = *scalars*	Channel number to be used to refer to each file in other statements (numbers for each type of file are independent); if this is set to a scalar containing a missing value, the first available channel of the specified type is opened and the scalar is set to the channel number
`FILETYPE` = *string tokens*	Type of each file (`input, output, unformatted, backingstore, procedurelibrary, graphics`); default `inpu`
`WIDTH` = *scalars*	Maximum width of a record in each file; default 80
`INDENTATION` = *scalar*	Number of spaces to leave at the start of each line; default 0
`PAGE` = *scalars*	Number of lines per page (relevant only for output files)
`ACCESS` = *string token*	Allowed type of access (`readonly, writeonly, both`); default `both`
`STYLE` = *string token*	Style in which to write to an output file (`plaintext`, `html`, `latex`, `rtf`); default `plai`
`HTMLHEAD` = *texts*	Text structures containing custom content for the header of an HTML document

The `OPEN` directive enables you to connect files to the various available channels within Genstat. Usually you need specify only the name of the file, the channel number and type of file, and leave the other parameters to take their default settings. For example, the following statements attach a file called `Weather.dat` to the second input channel, and then read data from it, as explained in 3.1.2.

```
OPEN 'Weather.dat'; CHANNEL=2; FILETYPE=input
READ [CHANNEL=2] Rain,Temperature,Sunshine
```

The filename can be anything that is acceptable to your computer system. You should, however, check for any constraints: for example, graphics software may require bitmap files to have the extension `.bmp`. You should check in your local documentation for information regarding any features that are specific to your computer or version of Genstat. For example, logical or symbolic names may be automatically translated by Genstat before files are accessed; upper and lower case characters may be significant, as on Unix systems. The filename may involve characters that have special meaning within Genstat. For example, the character `\` may be required to specify directories and sub-directories on a PC. As explained in 1.4.2, this character needs to be duplicated in a string to avoid Genstat interpreting it as the continuation symbol. For example

```
    OPEN 'D:\\UK\\Weather.dat'; CHANNEL=2; FILETYPE=input
```

to open the file `'D:\UK\Weather.dat'`. As a more convenient alternative, the Windows version of Genstat allows you to use `/` instead. Again, this should all be explained in the local documentation.

You are free to choose which channels you want to use (within the range available for the specified type of file), apart from input and output channel 1 which are "reserved" for use by the files specified on the command line. Also, input channel 5 is used for the start-up file (5.6.4) and, if you are working interactively, the standard start-up file arranges for output channel 5 to store a transcript of your output. However, you can use the `CLOSE` directive (3.3.2) to disconnect these files if you want to use the channels for some other purpose. The backing-store and unformatted work files are attached to channel 0, and this channel cannot be used in `OPEN` or `CLOSE`. Graphics files must be opened on the channel corresponding to the device number.

Obviously you cannot open more than one file on a channel, so if you wish to open a file on a channel that is currently in use you must first close that channel (3.3.2). Sometimes, in general programs or procedures, you may not know which channels are available. You can then let `OPEN` find a free channel: if `CHANNEL` is set to a scalar containing a missing value, the file is opened on the next available channel of the appropriate type, and the scalar is set to the number of the channel. The scalar need not be declared in advance; if `CHANNEL` is set to an undeclared structure, this will be defined as a scalar automatically.

```
    SCALAR FreeChan
    OPEN 'Weather.dat'; CHANNEL=FreeChan; FILETYPE=input
    READ [CHANNEL=FreeChan] Rain,Temperature,Sunshine
```

Another constraint is that you cannot open the same file on more than one channel at once.

Input files must already exist when they are opened, whereas output files will be created by Genstat. If an output file with the specified name exists already, Genstat may create an extra "version" of the file, or report a fault, or cause the file to be *overwritten*, depending on the usual conventions on your type of computer. Your local documentation will describe what rules apply in this situation, and should also explain if there are any system variables you can set to control this action.

The `STYLE` parameter controls the style to be used to represent the information in an output file. The default is to use plain text, which assumes that all characters occupy an equal width. So, for example, columns are aligned by use of space characters and captions are highlighted by underlining them by rows of equal signs or minuses. Alternatively, you can also choose a formatted style: HTML (as used for example by web browsers), RTF (as used by word processors such as Microsoft Word) or LaTeX (as used for scientific publications). Columns and tables are then formed using the conventions of the output format: for example, in LaTeX, they are written in the *tabular* environment. Note that, even if you have opened a channel in a formatted style, you can still switch to a plain-text mode, using the `OUTPUT` directive (3.4.3). Genstat then arranges to generate the output in an equally-spaced font, such as Courier.

When you open a file for use by backing store (3.5) or unformatted input and output (3.6), you can both read from it and send output to it, unless you set the `ACCESS` parameter (see below). Procedure libraries are a special type of backing-store file, described in 5.3.3.

The `WIDTH` parameter sets the maximum number of characters per line for input and output files. It is ignored for other types of file. The default values for `WIDTH` are designed to be appropriate for each implementation of Genstat and may differ between input and output; details will be found in your local documentation. For input and output with screen displays that use windows `WIDTH` may be set automatically from the size of the appropriate window.

For input files the default is normally 80, reflecting the size of most screen displays. You can change this if necessary, to read either fewer characters from each line, or longer lines. If the `WIDTH` is set to be too small any extra characters will be lost, which may cause unexpected action or syntax errors. Remember that if you use `READ` with `LAYOUT=fixed` to read fixed-format data, short lines are extended with spaces up to the `WIDTH` setting. If you want to read data from a file with, say, 64 characters per line, setting `WIDTH=64` when you open the file may make the format specification easier (rather than taking the default width of 80 and having to remember to skip 16 characters at the end of each line).

For output files, the default is the largest number of characters that can usually be displayed in a single line. This number is typically 80 for terminals but for files it is likely to be either 80, 120 or 132, depending on the type of computer. You can use the `WIDTH` parameter to restrict the number of output characters to a smaller number, or to a larger number up to 200.

The `PAGE` parameter specifies the size of page in output, affecting directives like `GRAPH`, `QUESTION` and `HELP`. For output to files, the default value of `PAGE` is designed to be suitable for printers. For windowed displays Genstat will, if possible, detect the size of the window and set the page size appropriately. You can also set option `OUTPRINT=page` in either `JOB` (5.1.1) or `SET` (5.6.1) to ensure that graphs and statistical analyses each start on a new page.

The `INDENTATION` parameter can be used to leave a specified number of blank characters to the left of each line of an output file, so that printed output can be bound for example. The indentation is subtracted from the `WIDTH` setting, so if you set `WIDTH=80` and `INDENTATION=10` then only 70 characters will be printed on each line of output.

The `ACCESS` parameter is used to control the way in which unformatted and backing-store files can be accessed, on computers that allow this; for details see your local documentation.

The `HTMLHEAD` parameter allows you to supply additional markup content for the document header of an HTML file, to be inserted between the `<head>` and `</head>` tags. It can be set either to a text containing all the HTML markup or to the name of a file containing that information. It is intended primarily for inserting CSS style information, for example:

```
<style>
h1 { color: black; background-color: red !important; }
h2 { color: white; background-color: green !important; }
</style>
```

but can also be used to set any other valid header content. Additional CSS content can also be loaded via a link tag, e.g.

```
<link rel="styleSheet" type="text/css" href="genstat.css">
```

By default, the header contains a title and some standard meta data. These tags can be overwritten by specifying these tags in the inserted header
data. The tags that are treated in this way are:

```
<title>
<meta name="description"
<meta name="keywords"
<meta name="author"
```

All other content of the text is inserted verbatim and assumed to be valid HTML. If `HTMLHEAD` is not set, Genstat inserts the content of the file `Genstat.css` which is supplied with the Genstat installation in the `Source` directory. This defines a number of classes which are used at various points in the Genstat output (for example to define styles used for output from `CAPTION`; see 3.2.3). The file can be used as a template from which to derive a local variation redefining basic elements of output.

3.3.2 The `CLOSE` directive

`CLOSE` directive

Closes files.

No options

Parameters

CHANNEL = *scalars* or *texts*	Numbers of the channels to which the files are attached, or identifiers of texts used for input (which, after "closing", can then be re-read)
FILETYPE = *string tokens*	Type of each file (input, output, unformatted, backingstore, procedurelibrary, graphics); default inpu
DELETE = *string tokens*	Whether to delete the file on closure (yes, no); default no

When you have finished with a file you can use CLOSE to release the channel to which it was attached, so that the channel is available for use with some other file. However, you do not need to close every file before you stop running Genstat; files are automatically closed at the end of every Genstat program.

Parameters CHANNEL and FILETYPE are similar to those of the OPEN directive. The DELETE parameter is useful if you are using files to store data temporarily, perhaps to release workspace within Genstat. When you have finished with the file you can set DELETE=yes to request that it be deleted on closure so that disk space is not wasted. For example,

```
OPEN 'Temp.bin'; CHANNEL=3; FILETYPE=unformatted
PRINT [CHANNEL=3;UNFORMATTED=yes] Surveys[1900,1910...1990]
DELETE Surveys[1900,1910...1990]

" ... and later on when you wish to retrieve the data ... "
READ [CHANNEL=3;UNFORMATTED=yes] Surveys[1900,1910...1990]
CLOSE 3; FILETYPE=unformatted; DELETE=yes
```

You cannot close a channel to which the terminal is attached, nor the current input or output channels. Also you cannot use CLOSE to delete files that have been opened with ACCESS=readonly or that are protected by the computer's file system.

3.3.3 The SKIP directive

SKIP directive

Skips lines in input or output files.

Options

CHANNEL = *scalar*	Channel number of file; default current channel of the specified type
FILETYPE = *string token*	Type of the file concerned (input, output); default inpu
STYLE = *string token*	Style to use when skipping output (plaintext, formatted); default * uses the current style of the channel

Parameter

identifiers	How many lines to skip; for input files, a text means skip until the contents of the text have been found, further input is then taken from the following line

This directive can be used for both input and output files. The FILETYPE and CHANNEL options indicate which file is to be skipped. By default this is the current input channel.

For input files you can skip over unwanted lines, which might be comments describing the data that is to follow, or might be some statements that you do not want to use in your current job. You can skip a specified number of lines, *n* say, by setting the parameter to a scalar containing the value *n*. Alternatively, you can skip everything up to and including a particular string of characters by setting the parameter to a text containing that string. For example,

```
    SKIP [CHANNEL=2] 'Section 2'
```

will skip the contents of the input file on channel 2 from the current position until the string `Section 2` is found. The next line to be read from channel 2 will then be the one immediately after the line containing `Section 2`.

For output files you can use `SKIP` to print blank lines to separate one section of output from another. You might want to do this if you had set the `PRINT` option `SQUASH=yes` (3.2.1) to suppress the automatic blank lines within a section of output. For example,

```
    PRINT [CHANNEL=2; IPRINT=*; SQUASH=yes] Heading
    SKIP  [CHANNEL=2; FILETYPE=output] 2
    PRINT [CHANNEL=2; IPRINT=*; SQUASH=yes] Table
```

places two blank lines between `Heading` and `Table` when printing their values to channel 2.

For an output file that has been opened in a style other than plain text (3.3.1), you can use the `STYLE` option to control whether the skipping is done in formatted or plain-text styles. If `STYLE` is not set, the default is to use the current style (as controlled by the `OUTPUT` directive; see 3.4.3).

3.3.4 The ENQUIRE directive

ENQUIRE directive

Provides details about files opened by Genstat.

No options

Parameters

`CHANNEL` = *scalars*	Channel numbers to enquire about; for `FILETYPE=input` or `output`, a scalar containing a missing value will be set to the number of the current channel of that type and a negative value can be used to check the existence of a file that is not yet connected to a channel
`FILETYPE` = *string tokens*	Type of each file (`input`, `output`, `unformatted`, `backingstore`, `procedurelibrary`, `graphics`); default `inpu`
`OPEN` = *scalars*	To indicate whether or not the corresponding channels are currently open (0=closed, 1=open)
`NAME` = *texts*	External name of the file, if channel is open
`EXIST` = *scalars*	To indicate whether files on corresponding channels currently exist (0=not yet created, 1=exist)
`WIDTH` = *scalars*	Maximum width of records in each file (only relevant for input and output files, set to `*` for other types)
`PAGE` = *scalars*	Number of lines per page (relevant only for output files)
`ACCESS` = *texts*	Allowed type of access: set to `'readonly'`, `'writeonly'` or `'both'`
`LINE` = *scalars*	Number of the current line (input files only)
`STYLE` = *texts*	Underlying style of an output channel: set to `'plaintext'`, `'html'`, `'rtf'`, or `'latex'`

`OUTSTYLE` = *texts*	Current style of an output channel: set to `'plaintext'` or `'formatted'`

`ENQUIRE` allows you to ascertain whether a particular channel is already in use and, if so, what properties are defined for aspects like the width of each line, the number of lines per page or the output style. This is likely to be of most use within general programs and procedures.

You specify the channel using the parameters `CHANNEL` and `FILETYPE`, in the usual way (3.3.1); the other parameters allow you to save the required information in data structures of the appropriate type. This is illustrated in Example 3.3.4.

Example 3.3.4

```
OPEN 'Weather.dat','Summary.out'; CHANNEL=2; FILETYPE=input,output
ENQUIRE 2,2; FILETYPE=input,output; \
  NAME=In2,Out2; WIDTH=InW2,OutW2; ACCESS=InAcc2,OutAcc2
PRINT In2,InW2,InAcc2,Out2,OutW2,OutAcc2; \
  FIELDWIDTH=20,7,10; DECIMALS=0; JUSTIFICATION=left

In2                 InW2   InAcc2    Out2                OutW2  OutAcc2
D:\UK\Weather.dat   80     readonly  D:\UK\Summary.out   80     writeonly
```

You can also use `ENQUIRE` to find out from within Genstat whether a file exists. You simply set the `CHANNEL` option to a negative number. For example,

```
ENQUIRE CHANNEL=-1; NAME='Lost.dat'; EXIST=Found
```

will set the scalar `Found` to one if the file `Lost.dat` exists, or to zero otherwise.

3.4 Managing input and output channels

Genstat always starts by processing statements from the keyboard or from the file attached to input channel 1. However, you can use the `INPUT` directive to change this within a job, to take statements from another file. Subsequently, you can use `INPUT` to switch to yet another channel, or `RETURN` to go back to the original file of statements. You can also use the `EXECUTE` directive (5.4.3) or macro substitution (1.8.2) to take input from a text structure containing Genstat statements. Similarly, any output from Genstat will be directed initially to output channel 1. When you start Genstat this will be connected either to the screen, in an interactive run, or to a file. You can change the current output channel at any time by using the `OUTPUT` directive (3.4.3). Also, if channel is in a formatted style (HTML, RTF or LaTeX; see 3.3.1), `OUTPUT` can switch it temporarily to the plain-text style (or back again). Once you have given an `OUTPUT` statement all output will appear in the file on this channel, until another `OUTPUT` statement is executed.

Many directives, like `PRINT`, have a `CHANNEL` option that lets you specify where the output from the directive is to go. This provides an alternative method of selectively diverting some output to a secondary file. You can also save complete transcripts of input or output in output files using the `COPY` directive (3.4.4).

3.4.1 Taking input statements from other files: the `INPUT` directive

`INPUT` directive

Specifies the input file from which to take further statements.

Options

`PRINT` = *string tokens*	What output to generate from the statements in the file

	(`statements`, `macros`, `procedures`, `unchanged`); default `stat`
`REWIND` = *string token*	Whether to rewind the file (`yes`, `no`); default `no`

Parameter

scalar	Channel number of input file

Having opened a file of Genstat statements on another input channel you can switch control to that channel at any time using an `INPUT` statement. You specify the channel as a number or as a scalar containing that number. For example,

```
OPEN 'Myprocs.gen'; CHANNEL=4; FILETYPE=input
INPUT 4
```

The file can contain any valid Genstat statements: they will be executed just as if they had been on the original input channel. In this file you could use an `INPUT` statement to switch back to channel 1 after a while. Alternatively, you may have set up several input files and jump from one to another, again using `INPUT`. You can use `RETURN` to go back to the previous channel or `STOP` to end this run of Genstat. If the end of the file is reached without finding any of these statements, control will be passed back to the previous input channel as described below in 3.4.2. Note that if you use `INPUT` to go back to an earlier channels you may affect the way in which `RETURN` works; details are given in 3.4.2.

The `PRINT` option can be used to specify whether the statements read from the file should be echoed to the current output channel. This is used in the same way as `INPRINT` in `JOB` (5.1.1) and `SET` (5.6.1).

The `REWIND` option allows you to return to the beginning of the file. You might need to do this, for example, if you had made an error, so that the statements on the secondary input file were executed wrongly. After correcting your error you could set `REWIND=yes` to start again from the beginning of the file.

3.4.2 The RETURN directive

RETURN directive

Returns to a previous input stream (text vector or input channel).

Options

`NTIMES` = *scalar*	Number of streams to ascend; default 1
`CLOSE` = *string token*	Whether to close the channel (or text) after the return (`yes`, `no`); default `no`
`DELETE` = *string token*	Whether to delete the text or the file to which the channel was attached (only relevant if `CLOSE=yes`) after the return (`yes`, `no`); default `no`

Parameter

expression	Logical expression controlling whether or not to return to the previous input stream; default 1 (i.e. true)

In its simplest form, you type

```
RETURN
```

to make Genstat stop taking statements from the current input channel and to go back to the channel that was previously active, and contained the `INPUT` statement that switched to the secondary file. Input then continues from the line following the original `INPUT` statement, but a marker is left in the channel that contains the `RETURN` statement, so that you can use `INPUT` to continue from the next line after `RETURN` later in your programme.

Sometimes you may want to return only if a particular condition is satisfied, for example if you have discovered that the data are unsatisfactory for whatever operations occur later in the file. To do this, you set the parameter to an appropriate logical expression; this must return a scalar result, which is interpreted as *true* if it is equal to 1, and *false* otherwise. For example

```
    RETURN MIN(Height)<0
```

If you have use `INPUT` several times, you may wish to return through several channels. The `NTIMES` option can be set to a number, or a scalar, to control how many returns take place. For example, with input starting on channel 1, supposing you had used `INPUT 2` to switch to a file on channel 2, and then `INPUT 3` to switch to a further file (on channel 3). If this file then contained the statement `RETURN [NTIMES=2]` you would return to channel 1. You can never return from input channel 1, so if you set `NTIMES` to a number greater than the number of currently active input channels, Genstat simply returns to channel 1.

You can set option `CLOSE=yes` to close the file after you have returned. Also, when `CLOSE=yes`, you can set option `DELETE=yes` to delete the file.

If Genstat meets the end of the file on the current input channel, it will try to return control to the channel from which it was called. This is called an *implicit return*. The channel is closed automatically when this happens, and a warning message is printed.

In order to maintain control over the different input channels, and know where to go after a `RETURN`, Genstat keeps an internal stack of input channels. Suppose you specify channel *k*, by typing `INPUT` *k*. There are three possible actions:

(a) if *k* is the current input channel, the statement is ignored;
(b) if *k* is not in the stack, it is added to it;
(c) if *k* is already in the stack (that is, the current state is: $1 \rightarrow ... \rightarrow k \rightarrow k_1 \rightarrow k_2 \rightarrow ... \rightarrow k_n$) then the intermediate channels k_1 ... k_n are suspended at their current positions and removed from the stack.

Input then switches to channel *k*, taking statements from the beginning of the file if it has never been used before, or from the point at which it was last suspended. Subsequent `INPUT` statements will re-start the other channels from where they were suspended. When a `RETURN` statement is used, Genstat steps back `NTIMES` through the stack, removing any intermediate channels from the stack. This means that, using the above representation of the input stack, if channel k_n contained the statement `INPUT` k_2 and channel k_2 then had a `RETURN`, this would return to channel k_1.

If you use `##` or `EXECUTE` to execute macros (1.8.2), these are treated in the same way as input channels and added to the input stack. You can use `INPUT` to temporarily halt a macro and switch to a file, and `RETURN` to get back to the macro.

3.4.3 Sending output to another file: the `OUTPUT` directive

`OUTPUT` directive

Defines where output is to be stored or displayed.

Options

`PRINT` = *string tokens*	Additions to output (`dots`, `page`, `unchanged`); default `dots,page`
`DIAGNOSTIC` = *string tokens*	What diagnostic printing is required (`messages`, `warnings`, `faults`, `extra`, `unchanged`); default `faul,mess,warn`
`WIDTH` = *scalar*	Limit on number of characters per record; default width

	of output file
INDENTATION = *scalar*	Number of spaces to leave at the start of each line; default 0
PAGE = *scalar*	Number of lines per page
STYLE = *string token*	Style for future output to the channel (plaintext, formatted); default * i.e. unchanged

Parameter

scalar	Channel number of output file

An OUTPUT statement changes the current output channel and thus re-defines where the output will be sent by the subsequent statements in a program, until another OUTPUT statement is given (excluding any statements that use a CHANNEL option to redirect their output). Thus

```
OUTPUT 2
PRINT X
PRINT [CHANNEL=3] Y
ANOVA X
```

sends the values of X, and the analysis of X by the ANOVA statement, to the file on the second output channel, and the values of Y to the file on the third.

The PRINT option controls two aspects of the output produced for example from statistical analyses: whether a line of dots is printed at the start, and whether the output begins on a new page; this can also be controlled by the OUTPRINT option of SET (5.6.1). Similarly, the DIAGNOSTIC option has exactly the same effect as the DIAGNOSTIC option of SET (5.6.1).

The WIDTH option specifies the maximum width to be used when producing output. The default value is the width specified when the file was opened (3.3.1), but you can subsequently decrease it; you cannot use OUTPUT to set the width to a greater value than that specified when the file was opened. The PAGE option allows you to reset the number of lines per page.

The STYLE option is relevant if the file on the channel has been opened in a style other than plain text. (The alternatives include HTML, RTF and LaTeX; see the OPEN directive, 3.3.1). It allows you to switch between the "formatted" style that is used by default for these files, and the ordinary plain-text representation. If the STYLE option is not specified, the style is left unchanged.

3.4.4 Saving a transcript of input or output: the COPY directive

COPY directive

Forms a transcript of a job.

Option

PRINT = *string tokens*	What to transcribe (statements, output); default stat

Parameter

scalar	Channel number of output file

The COPY directive can be used to save a copy of either input statements, or output, or both, in an output file. For example

```
OPEN 'Gen.rec','Gen.out'; CHANNEL=2,3; FILETYPE=output
COPY [PRINT=statements] 2
COPY [PRINT=output] 3
```

will keep a record of all the statements in the file Gen.rec and of all the output in the file

Gen.out. You can thus obtain output in more than one style (for example RTF and HTML as well as plain-text) by opening, and then copying, to files in the required styles (see 3.3.1).

Setting PRINT=* stops any copying to the specified channel. For example

```
COPY [PRINT=*] 2
```

stops copying to Gen.rec.

3.5 Storing and retrieving data structures

You will frequently want to save information that you have put into a data structure. This section explains how to transfer information to various other storage media on the computer, so that you can access the information easily later on.

There is an important difference between *storing* and merely printing. When you give the statement

```
PRINT [CHANNEL=2] X
```

you put only the identifier and the values of X into the character file attached to input channel 2. But if you give the statement

```
STORE [CHANNEL=2] X
```

you put all the details about X into the binary file attached to backing-store channel 2. So all the attributes of X are stored there too: for example, what type of structure it is, how long it is, and so on.

Section 3.5.1 describes the simplest use of storage and retrieval, which may be enough for most of your needs. Section 3.5.2 describes how backing-store files are arranged, with details of subfiles, userfiles and workfiles. Sections 3.5.3 to 3.5.6 describe the four directives that are relevant: STORE, RETRIEVE, CATALOGUE and MERGE.

3.5.1 Simple use of backing store

Here is an example to illustrate the simplest way of storing data, and then retrieving it. First, to store the scalar A and the variate B:

```
OPEN 'Example.gbs'; CHANNEL=1; FILETYPE=backingstore
SCALAR A; VALUE=2
VARIATE [VALUES=1...4] B
"Store structures A and B"
STORE [CHANNEL=1] A,B
```

The information about A and B is stored in the file named Example.gbs which is opened on backing-store channel 1 (3.3.1). There is actually an invisible intermediate stage here: A and B are first stored in a *subfile* by the STORE statement; this subfile is then stored in the userfile Example.gbs. The default name for the subfile is SUBFILE.

Example 3.5.1a shows how A can be retrieved in a subsequent job.

Example 3.5.1a

```
 2  OPEN 'Example.gbs'; CHANNEL=1; FILETYPE=backingstore
 3  " Retrieve structure A only "
 4  RETRIEVE [CHANNEL=1] A
 5  PRINT A

         A
     2.000
```

So far, the file consists of only one subfile, but you can add others if you want. To do this, you must give a subfile name:

```
OPEN 'Example.gbs'; CHANNEL=1; FILETYPE=backingstore
```

```
TEXT [VALUES='Storing more data','on backing store'] T
"Add new subfile called Newset to file"
STORE [CHANNEL=1; SUBFILE=Newset] T
```

There are now two subfiles in the file, called `SUBFILE` and `Newset`. Example 3.5.1b shows how to retrieve the text structure `T`.

Example 3.5.1b

```
   2  OPEN 'Example.gbs'; CHANNEL=1; FILETYPE=backingstore
   3  " Retrieve T and print it "
   4  RETRIEVE [CHANNEL=1; SUBFILE=Newset] T
   5  PRINT T

                  T
Storing more data
on backing store.
```

You can add as many new subfiles as you want, exactly as shown above, but you must keep the subfile names distinct within each file.

3.5.2 Subfiles, userfiles and workfiles

Before going any further, you need to know how structures are stored. A subfile is itself merely a portion of the backing-store file. Each subfile starts with a *catalogue*, recording which structures it stores. Then come the attributes (see 2.1) and the values of each structure. There are two types of subfiles. *Ordinary subfiles* can hold any type of structures except procedures; *procedure subfiles* hold only procedures (and their dependent structures).

Whenever you store a structure in a subfile, Genstat automatically stores also all the associated structures to which it points. If these associated structures also point to further structures, then they are stored too, and so on. Some of the structures may be unnamed (1.4.3) and some structures may be system structures (2.8). For example

```
TEXT [VALUES=A,B,C] T
FACTOR [LABELS=T; VALUES=1...3] F
STORE F
```

creates a subfile containing factor `F`. The complete definition of factor `F` depends on text `T` to supply level names. So `T` is stored too. The text `T` depends on a system structure (indicating the length of each line), which is therefore also stored. Hence to save factor `F`, Genstat has actually had to save three structures. However, this is all automatic, so you do not need to worry about any of the details of the system structures, and so on.

When you store a structure with a suffixed identifier, Genstat may have to set up a series of pointer structures if they are not already present (1.4.3 and 2.6). For example:

```
VARIATE [VALUES=1,2] V[1,2]
STORE [PRINT=catalogue] V[1]
```

The first line sets up a pointer structure `V`, pointing to `V[1]` and `V[2]`. To store variate `V[1]`, a pointer structure `V` has to be set up in the subfile, pointing to `V[1]` only. Thus two structures are saved on backing store, namely `V` and `V[1]`. The original pointer `V` in the program is left unchanged. (If the example had stored the whole of `V`, no such complications would have arisen.)

You can retrieve any pointer structure that you have set up in this way, and use it subsequently in the same way as any other pointer. But when a smaller pointer has to be set up only so that a suffixed identifier can be stored, no textual suffixes will be defined. So, if you want to store textual suffixes, you must define and store the pointer explicitly.

A backing-store file then consists of several subfiles; in fact the file can exist even if it is empty. However, if a file containing anything that has not been stored by a Genstat backing-store statement is attached as a backing-store file, it will be rejected.

A file that can be read by another job is called a *userfile*; it is permanent, in the sense that it will continue to exist after you have finished the job that created it.

Each job can have one temporary file called the *backing-store workfile* which also consists of a set of subfiles. The workfile's catalogue is deleted at the end of each job. The workfile itself may be overwritten in a later job in the same Genstat program, and on most computers it will be deleted automatically by `STOP`. However, if you abandon a run of Genstat before it has ended (for example by rebooting a PC), the workfile may survive.

A subfile name can be either an unsuffixed identifier or a suffixed identifier (1.5.3) with a numerical suffix. The identifiers of subfiles are kept in a separate catalogue to the identifiers of data structures, so you do not need to worry about keeping the identifiers data structures and subfile distinct. However, if you use a suffixed identifier for a subfile, `Sub[1]` say, you cannot also use the identifier `Sub`.

3.5.3 The `STORE` directive

`STORE` directive

To store structures in a subfile of a backing-store file.

Options

`PRINT` = *string token*	What to print (catalogue); default `*`
`CHANNEL` = *scalar*	Channel number of the backing-store file where the subfile is to be stored; default 0, i.e. the workfile
`SUBFILE` = *identifier*	Identifier of the subfile; default `SUBFILE`
`LIST` = *string token*	How to interpret the list of structures (inclusive, exclusive, all); default `incl`
`METHOD` = *string token*	How to append the subfile to the file (`add`, `overwrite`, `replace`, `update`); default `add`, i.e. clashes in subfile identifiers cause a fault (note: replace overwrites the complete file)
`PASSWORD` = *text*	Password to be stored with the file; default `*`
`PROCEDURE` = *string token*	Whether subfile contains procedures only (`yes`, `no`); default `no`
`UNNAMED` = *string token*	Whether to list unnamed structures (`yes`, `no`); default `no`
`MERGE` = *string token*	Whether or not to merge the structures with the existing contents of the subfile (`yes`, `no`); default `no`

Parameters

`IDENTIFIER` = *identifiers*	Identifiers of the structures to be stored
`STOREDIDENTIFIER` = *identifiers*	Identifier to be used for each structure when it is stored

The structures to be stored are specified by the `IDENTIFIER` parameter. The `CHANNEL` option indicates the backing-store file to use, and the `SUBFILE` option specifies the subfile that is created. Both these options can be omitted; by default the file will be the workfile, and the subfile will be called `SUBFILE`. The structures that are stored in the subfile are merely copies of the structures in the job, so the original structures remain available for further use within the job.

The `STOREDIDENTIFIER` parameter allows you to give a structure a different name within the subfile: For example,

```
VARIATE [VALUES=10.2,15.3,21.4,16.8,22.3] Weight
STORE Weight; STOREDIDENTIFIER=WtWeek2
```

stores a structure with identifier `Weight` within Genstat as a structure with identifier `WtWeek2` in the backing-store file. If you want to rename only some of the structures, you can either respecify the existing identifier, or insert `*` at the appropriate point in the list. For example, you could store `X` and `Y`, renaming only `Y` as `Yy`, by

```
    STORE X,Y; STOREDIDENTIFIER=X,Yy
```

or by

```
    STORE X,Y; STOREDIDENTIFIER=*,Yy
```

You can give an unnamed structure in the list of either parameter. For example

```
    STORE !(10.2,15.3,21.4,16.8,22.3); STOREDIDENTIFIER=WtWeek2
```

But of course you will not be able to retrieve any structure that has been stored as an unnamed structure (except perhaps as a dependent structure of another structure, see 3.5.2).

All the structures in a subfile must have distinct identifiers, and Genstat will report a fault if you try to give two the same name. You thus need to be careful if you are storing structures inside a procedure, as the same identifier can be used for one structure within the procedure, and for another one outside; you cannot store both in the same subfile.

Procedures that have been retrieved automatically from libraries (5.3.3) cannot be stored by `STORE`.

You can set option `PRINT=catalogue` to obtain a catalogues of the subfiles in the backing-store file, and of the structures in the subfile just created. If you also set option `UNNAMED=yes` Genstat will also list any unnamed structures, with details of how they depend on each other.

The `LIST` option controls how the `IDENTIFIER` list is interpreted. The default setting `inclusive` simply stores the structures that have been listed.

Alternatively, if you set `LIST=all` Genstat will store all the structures in the current job that have identifiers and whose types have been defined. If the statement is inside a procedure, then only the structures defined within the procedure are stored (5.3). If you are storing procedures, then this setting will store all procedures that you have created explicitly in this job, by `PROCEDURE` or `RETRIEVE` statements.

Finally, you can set `LIST=exclusive` to store everything that you have not included in the `IDENTIFIER` parameter: that is, all the other named structures that are currently accessible, or all the other procedures that have been created in this job. Note, though, that some of the structures in the `IDENTIFIER` list may be stored if they are needed to complete the set of structures to be stored. If you use this setting, the `STOREDIDENTIFIER` parameter is ignored. For example

```
    TEXT [VALUES=a,b] T
    FACTOR [LABELS=T] F
    TEXT [VALUES='variate  text'] Vt
    VARIATE V; EXTRA=Vt
```

creates four named structures, `T`, `F`, `V` and `Vt`. The statement

```
    STORE [LIST=inclusive] T
```

stores the text `T`;

```
    STORE [LIST=all]
```

stores all the four structures that have identifiers;

```
    STORE [LIST=exclusive] F,T
```

stores `Vt` and `V`; and

```
    STORE [LIST=exclusive] Vt,T
```

results in all four structures being saved, because `V` points to `Vt`, and `F` points to `T`.

If a subfile of the specified name already exists on the backing-store file, the storing operation will usually fail. You can then set option `METHOD=overwrite` to overwrite the old subfile, that is, to replace the old subfile with a new subfile; alternatively, you can put `METHOD=replace` to form a new backing-store file containing only the new subfile. Setting `METHOD=update` adds new structures to an existing subfile. The `MERGE` option then controls what happens if a data structure that is being added to the file is already present; by default it overwrites the previous version but, if you put `MERGE=yes`, only new structures are added to the file.

To make your files secure, you can specify a password using the `PASSWORD` option. Once you have done this, you must include the same password in any future use of `STORE` or `MERGE` with this same userfile; spaces, case, and new lines are significant in the password. You cannot change the password in a userfile once you have set it, but you can use the `MERGE` directive to create a new userfile with no password or with a new password. If you set the password to be a text whose values have been restricted (4.4.1), the restriction is ignored.

The `PROCEDURE` option indicates whether the subfile is to store procedures (`PROCEDURE=yes`), or ordinary data structures.

3.5.4 The RETRIEVE directive

RETRIEVE directive

Retrieves structures from a subfile.

Options

`CHANNEL` = *scalar*	Specifies the channel number of the backing-store or procedure-library file containing the subfile (`FILETYPE` settings `'back'` or `'proc'`); default 0 (i.e. the workfile) for `FILETYPE=back`, no default for `FILETYPE=proc`, not relevant with other `FILETYPE` settings
`SUBFILE` = *identifier*	Identifier of the subfile; default `SUBFILE`
`LIST` = *string token*	How to interpret the list of structures (`inclusive`, `exclusive`, `all`); default `incl`
`MERGE` = *string token*	Whether to merge structures with those already in the job (`yes`, `no`); default `no`, i.e. a structure whose identifier is already in the job overwrites the existing one, unless it has a different type
`FILETYPE` = *string token*	Indicates the type of file from which the information is to be retrieved (`backingstore`, `procedurelibrary`, `siteprocedurelibrary`, `Genstatprocedurelibrary`); default `back`

Parameters

`IDENTIFIER` = *identifiers*	Identifiers to be used for the structures after they have been retrieved
`STOREDIDENTIFIER` = *identifiers*	Identifier under which each structure was stored

---

You recover information from a subfile of a backing-store file using the `RETRIEVE` directive. The `CHANNEL` option specifies the backing-store file, and the `SUBFILE` option indicates the subfile. Both these options can be omitted; by default the file will be the workfile, and the subfile will be called `SUBFILE`.

When you retrieve a structure Genstat may also retrieve a chain of associated structures: that is, all the structures to which it points, and the structures to which they point, and so on. For example, suppose you store the three structures with identifiers `T`, `V` and `F`, along with an unnamed structure storing information about `T`, in a subfile called `SUBFILE` in backing-store file `File1.gbs`:

```
    OPEN 'File1.gbs'; CHANNEL=1; FILETYPE=backingstore
    TEXT [VALUES=a,b,c] T
    VARIATE V; EXTRA=T
    FACTOR [LABELS=T] F
    STORE [CHANNEL=1] T,V,F
```

Then the statement

```
    RETRIEVE [CHANNEL=1] V
```

will retrieve not only V but also T (which was associated with T by the EXTRA parameter of the VARIATE statement), and the unnamed structure that is associated with T. The structures V, T and the unnamed structure, are said to be a *complete set* from the subfile.

The IDENTIFIER parameter specifies the structures to be retrieved. You can use the STOREDIDENTIFIER parameter to give a structure a different name from the one within the subfile. For example

```
    RETRIEVE IDENTIFIER=Weeks; STOREDIDENTIFIER=Time
```

You are not allowed to give identical identifiers to two retrieved structures, nor are you allowed to have the same identifier referring to a structure of one type in a subfile, and to a structure of a different type in your job.

As with STORE, if you want to rename only some of the structures, you can either respecify the existing identifier, or insert * at the appropriate point in the STOREDIDENTIFIER list.

Genstat knows whether you are retrieving a procedure by the type of SUBFILE that you are accessing. You are not allowed to rename a procedure as a suffixed identifier or as the name of a directive.

You can even rename a structure so that it is unnamed in the job. Suppose, for example, that a structure T already exists within Genstat, and that you want to retrieve the variate V stored in the file File1.gbs above. Then, as we have seen, the structure T will also be retrieved. However, you can avoid the existing structure T job being overwritten by making the retrieved version of T unnamed:

```
    OPEN 'File1.gbs'; CHANNEL=1; FILETYPE=backingstore
    RETRIEVE [CHANNEL=1] V,!T(a); STOREDIDENTIFIER=V,T
```

The value, a, of the unnamed text !T(a) will be replaced by the values stored for T, and this unnamed text will become the EXTRA text for V. Alternatively you could rename T to be Tnew by

```
    RETRIEVE [CHANNEL=1] V,TNew; STOREDIDENTIFIER=V,T
```

When you are retrieving a suffixed identifier, Genstat matches the numerical suffix only, and not the whole structure that is denoted by the identifier. For example, suppose pointer P stored in a subfile points to structures with identifiers A, B, C and D, and that P has numerical suffixes 1 to 4 respectively. Also suppose that in your current job, you have never mentioned pointer P either directly or indirectly. Then the statement

```
    RETRIEVE [CHANNEL=1] P[2]
```

will retrieve the structure B from backing store but, as it has not been referenced only as P[2] in the RETRIEVE statement, the identifier B will not be recovered and it will be known only as P[2] within Genstat.

A structure that you are retrieving from a subfile may sometimes overwrite the values of an existing structure in your program. If this structure is a pointer or a compound structure, the existing suffixes will be overwritten by those of the stored structure, so some existing structures with suffixed identifiers may in effect be lost. For example, suppose that userfile File2.gbs contains a pointer P, with suffixes 1 and 2 pointing to structures A and B. If we set up a variate P[3], and then retrieve the pointer P

```
    OPEN 'File2.gbs'; CHANNEL=1; FILETYPE=backingstore
    VARIATE [VALUES=1...6] P[5,6,7]
    RETRIEVE [CHANNEL=1] P
```

P will now have suffixes 1 and 2 pointing to A and B, but the variate P[3] will have been lost. For more details about pointers, see 2.6.

The LIST option is similar to its namesake in the STORE directive (3.5.3), but it now refers to the named structures in the subfile.

The FILETYPE option specifies whether you wish to retrieve information from backing-store files that have been attached as normal backing store files or as procedure libraries by the OPEN directive (3.3.1), or from the Genstat Procedure library or from the site procedure library. The CHANNEL setting is ignored if the siteprocedurelibrary or Genstatprocedurelibrary settings are used. The source code of the procedures in the Genstat Procedure library can be accessed more easily using the LIBEXAMPLE procedure or, in Genstat *for Windows*, by selecting Procedure Source from the Help menu on the menu bar.

Normally when you retrieve a complete subset of structures, Genstat overwrites all structures in the job that have the same identifier (after any renaming). As a result, some other structures already in the job may become inconsistent, and will be destroyed. You can avoid this by setting option MERGE=yes. Then Genstat does not overwrite any structures with the same name and type. A consequence, however, is that some of the retrieved structures may now be inconsistent, and thus need to be destroyed in the program (although they will of course remain in the subfile).

3.5.5 The CATALOGUE directive

CATALOGUE directive

Displays the contents of a backing-store file.

Options

PRINT = *string tokens*	What to print (subfiles, structures); default subf, stru
CHANNEL = *scalar*	Channel number of the backing-store file; default 0, i.e. the workfile
LIST = *string token*	How to interpret the list of subfiles (inclusive, exclusive, all); default incl
SAVESUBFILE = *text*	To save the subfile identifiers; default *
UNNAMED = *string token*	Whether to list unnamed structures (yes, no); default no

Parameters

SUBFILE = *identifiers*	Identifiers of subfiles in the file to be catalogued
SAVESTRUCTURE = *texts*	To save the identifiers of the structures in each subfile

You can use CATALOGUE to obtain details of the subfiles contained in a backing-store file, or the structures within an ordinary subfile, or the procedures within a procedure subfile. The file is indicated by the CHANNEL option, and the SUBFILE parameter specifies the subfiles (of ordinary structures or of procedures) that are to be catalogued.

The PRINT option specifies which catalogues are to be printed. The subfiles setting prints the catalogue of subfiles in the backing-store file attached to the channel specified by the CHANNEL option, while the structures setting prints the catalogue of structures or procedures that are in the subfiles specified by the SUBFILE parameter.

If you set option UNNAMED=yes the unnamed structures in each subfile will also be listed, together with details of how the structures depend on each other.

The LIST option is similar to its namesake in the STORE directive, but it now refers to the identifiers in the SUBFILE list.

The `SAVESTRUCTURE` parameter allows you to set up texts, one for each subfile in the `SUBFILE` parameter. Each text contains the identifiers of all structures with an unsuffixed identifier in the subfile. Each identifier is put on a separate line, and the characters `,\` are appended to all but the last line. You would normally use these texts as a macro; the `,\` makes them useable as lists of identifiers. If the text is used as a macro, it is subject to the restriction on the length of statements (1.7). The `SAVESUBFILE` option allows you to save a similar text containing the identifiers of all the subfiles in a backing-store file.

Suppose we have a userfile called `File3.gbs` which already contains three subfiles: two ordinary subfiles (called `Sub[1]` and `Sub[2]`), and a procedure subfile called `Sub3` containing a procedure called `%TRANSFORM`. Example 3.5.5a adds a fourth subfile called `Sub4`.

Example 3.5.5a

```
TEXT [VALUES='vector_1','vector_3'] L
& [VALUES='heading'] T
POINTER [SUFFIXES=!(1,3); NVALUES=L] P
VARIATE P[],V; EXTRA=T
OPEN 'File3.gbs'; CHANNEL=1; FILETYPE=backingstore
STORE [CHANNEL=1; SUBFILE=Sub4] P,V
CATALOGUE [PRINT=structures; CHANNEL=1] Sub4

Catalogue
=========

catalogue of structures in the subfile Sub4

    entry  identifier     type
    1      P              pointer
    2      V              variate
    3      P[1]           variate
    4      P[3]           variate
    5      T              text
```

The subfile contains five named structures with identifiers `P`, `V`, `P[1]`, `P[2]` and `T`. There are also two unnamed structures associated with `P` and `T`, as can be seen when we set option `UNNAMED=yes` in Example 3.5.5b, and obtain a more detailed catalogue. This also gives details of any dependencies among structures, referenced by their index in the entry column. The identifier column gives the numerical suffix, and the labels column gives textual suffixes. This information is particularly helpful with complicated trees of pointers (2.6) or with compound structures (2.7).

Example 3.5.5b

```
CATALOGUE [PRINT=Structures; CHANNEL=1; UNNAMED=Yes] \
  Sub4; SAVESTRUCTURE=Tsub4

Catalogue
=========

Catalogue of structures in the subfile Sub4

    entry  identifier     type        points to
    1      P              pointer     3
                                      pointer values
                                      unit            entry    labels
                                      1               4        vector_1
                                      3               5        vector_3
    2      V              variate     6
    3                     text        7
    4      P[1]           variate     6
    5      P[3]           variate     6
    6      T              text        8
    7                     system
    8                     system

  11  PRINT Tsub4; JUSTIFICATION=left

Tsub4
P,\
V,\
T
```

The text `Tsub4`, saved by the `SAVESTRUCTURE` parameter of `CATALOGUE`, gives an identifier list of all the structures in the subfile that can be accessed directly.

Example 3.5.5c produces a catalogue of the procedure subfile `Tsub3` (line 12), and then produces a catalogue of the subfiles (line 13), at the same time using the `SAVESUBFILE` option to place the subfile names into the text `Tsubf`. Notice that Tsubf excludes the system-type subfile Sub, which exists because two of the subfiles (Sub[1] and Sub[2]) have suffixed identifiers, and cannot be used as a subfile in its own right.

Example 3.5.5c

```
  12  CATALOGUE [PRINT=structures; CHANNEL=1] Sub3

Catalogue
=========

Catalogue of procedures in the procedure subfile Sub3

    entry  identifier
    1      %TRANSFORM

  13  CATALOGUE [PRINT=subfiles; CHANNEL=1; SAVESUBFILE=Tsubf]

Catalogue
=========

Catalogue of subfiles in the userfile 1

    entry  identifier     type
    1      Sub4           ordinary
    2      Sub3           procedure
    3      Sub            system
    4      Sub[2]         ordinary
    5      Sub[1]         ordinary

  14  PRINT Tsubf; JUSTIFICATION=left

Tsubf
Sub4,\
Sub3,\
Sub[2],\
Sub[1]
```

3.5.6 The `MERGE` directive

`MERGE` directive

Copies subfiles from backing-store files into a single file.

Options

`PRINT` = *string token*	What to print (catalogue); default `*`
`OUTCHANNEL` = *scalar*	Channel number of the backing-store file where the subfiles are to be stored; default 0, i.e. the workfile

METHOD = *string token*	How to append subfiles to the OUT file (add, overwrite, replace); default add, i.e. clashes in subfile identifiers cause a fault (note: replace overwrites the complete file)
PASSWORD = *text*	Password to be checked against that stored with the file; default *

Parameters

SUBFILE = *identifiers*	Identifiers of the subfiles
INCHANNEL = *scalars*	Channel number of the backing-store file containing each subfile
NEWSUBFILE = *identifiers*	Identifier to be used for each subfile in the new file

The MERGE directive copies subfiles into another backing-store file. You can either add the subfiles to an existing backing-store file, or form a new backing-store file.

The OUTCHANNEL option specifies the backing-store channel of the file to which the subfiles are to be copied; by default this is the workfile (channel 0).

The SUBFILE parameter specifies the list of subfiles that are to be copied, and the INCHANNEL parameter indicates the channel of the backing-store file where each one is currently stored. If you do not specify the INCHANNEL parameter, Genstat assumes that the subfiles are coming from the workfile. You are not allowed to include the OUTCHANNEL among the channels in the INCHANNEL list. Also, you cannot store two subfiles with the same names, and should use the NEWSUBFILE parameter to rename any that clash. For example

```
MERGE [OUTCHANNEL=3] JanData,JulyData,JanData; \
  INCHANNEL=1,1,2; NEWSUBFILE=Jan92dat,Jul92dat,Jan93dat
```

To rename only some of the subfiles, you can either respecify the existing identifier, or insert * at the appropriate point in the NEWSUBFILE list.

If you specify a missing identifier * in the SUBFILE list, Genstat will include all the subfiles from the relevant INCHANNEL. If you want to rename any of these subfiles, you can also mention it explicitly. For example, this statement will take all the subfiles from channel 1 and rename subfile Sub as Subf.

```
MERGE *,Sub; INCHANNEL=1; NEWSUBFILE=*,Subf
```

You can set option PRINT=catalogue to produce a catalogue of the subfiles in the new backing-store file (3.5.5).

If a subfile of the specified name already exists on the backing-store file, the storing operation will usually fail. However, you can set option METHOD=overwrite to overwrite the old subfile, that is, to replace the old subfile with a new subfile. Alternatively, you can put METHOD=replace to form a new backing-store file containing only the new subfiles.

Subfiles are merged in a fixed order. Genstat first takes the subfiles from the backing-store file with the lowest channel number, in the order in which they occur there, then it takes the subfiles the next lowest channel number, and so on. If OUTCHANNEL=0 (that is, the new file is the workfile), the original subfiles that are to be retained from that file will be followed by the new subfiles; otherwise, if OUTCHANNEL is non-zero, the original subfiles are placed after the new subfiles. If you want to put the subfiles into a particular order, you should merge them into the workfile in that order, and then merge the workfile into a new userfile.

To keep the new file secure, you can use the PASSWORD option to incorporate a password, as explained in 3.5.3.

3.6 Storing and retrieving programs and data in unformatted files

The `RECORD` directive (3.6.1) allows you to produce a backing-store file containing all the details required to recreate the current state of a Genstat job. You can then use the `RESUME` directive (3.6.2), either later in your program, or during a completely different Genstat run, to recover all this information and continue your use of Genstat from that point. This can be useful if you need to abandon an analysis and resume it at some later date, or if you want to save the current state of a program in case your next operations turn out to be unsuccessful.

3.6.1 The RECORD directive

RECORD directive

Dumps a job so that it can later be restarted by a `RESUME` statement.

Option

`CHANNEL` = *scalar*	Channel number of the backing-store file where information is to be dumped; default 1

No parameters

`RECORD` sends all the relevant information about the current state of your Genstat job to the backing-store file specified by the `CHANNEL` option. You can then use the `RESUME` directive to re-establish that situation either in a future Genstat run, or later in the same run.

`RECORD` stores all the data structures in the file, using the equivalent of the statement

```
STORE [LIST=all]
```

and then adds a private section, at the end of the file, to store all the remaining information. The file can thus be used with the `RETRIEVE` directive in the ordinary way, to recover individual data structures without having to recover the whole job. However, if you add extra data structures (using `STORE`) to a file from `RECORD`, or merge it with other backing-store files (using `MERGE`), the additional information is lost and the result is an ordinary backing-store file.

The information includes the current graphics settings, any current setting of the `UNITS` directive and settings of model-definition directives (`MODEL`, `BLOCKSTRUCTURE`, `TREATMENTSTRUCTURE`, `COVARIATES`, `VCOMPONENTS`, `VSTRUCTURE`, and so on), but no details are kept of the files that are open on any of the channels. If you use `RECORD` with the same file again, the earlier information is overwritten.

3.6.2 The RESUME directive

RESUME directive

Restarts a recorded job.

Options

`CHANNEL` = *scalar*	Channel number of the backing-store file where the information was dumped; default 1
`CLOSE` = *string token*	Whether to close the file afterwards (`yes`, `no`); default `no`

No parameters

`RESUME` recovers the information stored by a previous `RECORD` statement so that you can continue your use of Genstat as though nothing had happened in between. Thus, for example, Genstat deletes all the data structures that were created in the current job prior to `RESUME`, and reinstates the data structures that were available in Genstat at the time the `RECORD` statement took place. Similarly, the current graphics settings are replaced by those that were in force when `RECORD` was used, but any external files that are attached to Genstat remain unaffected.

If the `RECORD` directive was used within a procedure or a `FOR` loop, the job is not resumed at that point. Instead, it restarts at the statement after the procedure call, or after the outermost `ENDFOR` statement.

The `CHANNEL` option specifies the channel to which the file has been connected (this can be done using the `OPEN` directive). You can set the `CLOSE` option to `yes` to close the file after the information has been recovered.

Example 3.6.2 illustrates how `RECORD` and `RESUME` are used.

Example 3.6.2

```
   2  OPEN 'DUMP.GBS'; CHANNEL=1; FILETYPE=backingstore
   3  VARIATE [VALUES=1,2] A
   4  RECORD
   5  PRINT A

          A
      1.000
      2.000

   6  CLOSE 1; FILETYPE=backingstore
   7  ENDJOB

********* End of job.

Genstat 64-bit Release 19.1  (PC/Windows 8)      19 October 2016 10:12:48
Copyright 2016, VSN International Ltd.
Registered to: VSNi

   8  OPEN 'DUMP.GBS'; CHANNEL=1; FILETYPE=backingstore
   9  RESUME
  10  CALCULATE A = A+1
  11  PRINT A

          A
      2.000
      3.000
```

3.7 Storing and reading data with unformatted files

Unformatted files can be used to store values of data structures using `PRINT` (3.2.1), so that they can later be input again using `READ` (3.1.1). This provides a convenient way to free some space temporarily. It can also save computing time if you have a large data set that may need to be read several times. Input from character files is slow. So, after vetting a large data set, it will be read more efficiently on future occasions if you transfer its contents to an unformatted file. As an alternative you could use backing store, but this stores the attributes of the structures as well as their values, and so access will take longer. You can also use these facilities to transfer data between Genstat and other programs.

Unformatted files are selected in `READ` and `PRINT` by setting option `UNFORMATTED=yes`. The only options that are then relevant are `CHANNEL`, `REWIND` and `SERIAL`.

Genstat automatically creates an *unformatted workfile*, on channel 0, to which unformatted output is sent by default (by `PRINT`), and from which unformatted input is taken by default (by `READ`). This file is deleted automatically by the `STOP` directive.

It is usually quicker to read and write structures in series. Also, the values of the structures transferred in parallel must all be of the same *mode*. Neither texts nor factors can be stored in parallel with values of the other, numerical, structures: scalars, variates, matrices or tables. As an example, we first open a file, and declare some variates, matrices and factors.

```
OPEN 'Bdat'; CHANNEL=3; FILETYPE=unformatted
VARIATE X,Y,Z; VALUES=!(11...19),!(21...29),!(31...39)
MATRIX [ROWS=2; COLUMNS=3; VALUES=11,12,13,21,22,23] M
FACTOR [LEVELS=3; VALUES=1,3,2,3,1,2,2,2,1,3] F
```

The next three statements store data for `M` and `F` on the file named `BDAT` and data for `X`, `Y` and `Z` (in parallel) on the workfile.

```
PRINT [CHANNEL=3; SERIAL=yes; UNFORMATTED=yes] M,F
PRINT [UNFORMATTED=yes] X,Y,Z
```

You can now free the space for numerical data for other purposes, by putting

```
DELETE X,Y,Z,F,M
```

By rewinding the files we can read the data back into Genstat.

```
READ [UNFORMATTED=yes; REWIND=yes] X,Y,Z
READ [CHANNEL=3; SERIAL=yes; UNFORMATTED=yes; REWIND=yes] M,F
```

You can also re-use the external file `BDAT` in a later job.

If you change the lengths of structures, you must remember to reset them to their original values before you use unformatted `READ` to recover the data values from the file. Only the data values are stored in unformatted files, and not the attributes (such as lengths) as in backing-store files.

3.8 Input and output from other systems

Genstat *for Windows* has several specialized commands for exchanging data with other other systems, such as databases, spreadsheets and other statistical systems. You can also read and write files in Genstat's own spreadsheet format. Details are in the on-line help. These procedures may not be present, however, in some other implementations.

`IMPORT`	reads data in a foreign file format, loads it or converts it to a Genstat spreadsheet file; supported file types include Excel 2-5,95,97,2000,XP,2003, Lotus, Quattro, dBase 2-5, Paradox 3-9, SAS PC 6.03-12, 7-9, SAS Transport, SAS JMP, Minitab 8-13, Statistica 5 and 6, Systat, MStat, Instat, Epi-Info, SPSS/Win, Gauss Data/Matrix (PC/Win/Unix), MatLab, Splus (PC/Unix), Stata 4-8, R data frames, Weka Attribute files, SigmaPlot 7-9, OSIRIS, Limdep, comma delimited text files, ArcView/Info Shapefiles, MapInfo Exchange files, Windows Bitmap, Windows Sound and NMR Binary files
`EXPORT`	saves data to Excel, Quattro, dBase, SPlus, Gauss, MatLab or Instat
`SPLOAD`	loads a Genstat spreadsheet file
`CSPRO`	reads a data set from a CSPro survey data file and dictionary, loads it into Genstat or puts it into a spreadsheet file
`FSPREADSHEET`	creates a Genstat Spreadsheet file
`DBCOMMAND`	runs an SQL command on an ODBC database
`DBIMPORT`	loads data from an ODBC database
`DBEXPORT`	updates an ODBC database table from Genstat
`DDEIMPORT`	gets data from a Dynamic Data Exchange (DDE) server
`DDEEXPORT`	sends data or commands to a DDE server

4 Calculations and data manipulation

Genstat has many directives for doing calculations or for manipulating data, and a full range of mathematical and statistical functions (4.2). There is also a directive to link to algorithms in the Numerical Algorithms Group (NAG) Library (4.13). Other facilities are provided by procedures, mainly in the `Manipulation` module of the procedure library.

You may wish to use these facilities as part of a statistical analysis; for example, you may want to transform your data before fitting a regression or doing an analysis of variance. However, they can be useful even if you have no intention of doing a statistical analysis but merely wish to use Genstat as a package for data-handling and arithmetic, or as a mathematical tool e.g. for systems modelling. This introduction mentions all the relevant directives and procedures, giving section references for the more important commands, which are described in this manual. Information about the other commands can be found in the *Genstat Reference Manual*.

The `CALCULATE` directive (4.1) allows you to perform straightforward arithmetic operations on any numerical data structure. It also enables you to make logical tests on data: for example, you may want to check whether two variates contain the same values; similar checks can be done with factors, texts and pointers. You can use `CALCULATE` for matrix operations: for example, matrix multiplication, inversion and Choleski decompositions (4.1.3 and 4.2.4). `CALCULATE` can do calculations with tables, and these need not have identical sets of classifying factors (4.1.4). When you use `CALCULATE`, the results are stored in appropriate data structures (which may be defined for you automatically: 4.1.5).

`CALCULATE`	performs arithmetic and logical calculations (4.1)

If you want to use the results only once, do not forget that you can use an expression anywhere that Genstat expects a list of identifiers (1.5.3). The rules for defining these expressions are exactly as explained below for `CALCULATE`. Knowledge of the rules may also provide useful background information to the Calculate menu of Genstat *for Windows* (which uses `CALCULATE`). This menu allows you to assemble the calculation by selecting data structures from an Available Data window, and clicking appropriate buttons to select the various operators (addition, multiplication and so on).

Another very general and powerful directive is `EQUATE`, which allows values to be copied from one set of data structures to another; the structures must store values of the same mode (for example, numbers or text), but need not be of the same type. For example, you may want to copy the columns of a matrix into a list of variates. The `SETRELATE` directive allows you to compare the sets of values in two different structures with values of the same mode (but not necessarily of the same type), `SETCALCULATE` performs calculations on sets, and `SETALLOCATIONS` allows you to form all the ways in which a set of objects can be allocated to subsets.

`EQUATE`	copies values between sets of data structures (4.3.1)
`SETRELATE`	compares the sets of values in two data structures (4.3.2)
`SETCALCULATE`	performs Boolean set calculations on the contents of vectors and pointers (4.3.3)
`SETALLOCATIONS`	runs through all ways of allocating a set of objects to subsets with specified sizes (4.3.4)

There are several commands for manipulating vectors (variates, factors or texts). A "restriction" can be associated with a vector, so that subsequent statements operate on only a subset of its units. Alternatively, you may wish to store the subset, in a data structure on its own. Units of vectors can be sorted into systematic order or into random order, and you can select random samples of a set of units. You can form a vector containing the values of a set of vectors of the same type, appended together, along with a factor which indicates the vector from which each unit came. Similarly, data matrices can be combined by "stacking" (or appending) their corresponding vectors. Another type of combination is to "join" (or merge) new vectors into a data matrix according to the values of one or more "key" vectors. You can also form a set of variates, each of which contains the values from one of the units of every member of a set of structures.

`RESTRICT`	defines a "restriction" on the units of a vector (4.4.1)
`SUBSET`	forms vectors containing subsets of the values in other vectors (4.4.2)
`FREGULAR`	expands vectors onto a regular two-dimensional grid (procedure)
`FRESTRICTEDSET`	forms vectors with the restricted subset of a list of vectors (procedure)
`SORT`	sorts units of vectors into alphabetic or numerical order of an index vector, or forms a factor from a variate or text (4.4.3)
`RANDOMIZE`	puts the units of a set of vectors into random order, or randomizes the units of an experimental design (2:4.11.1)
`SAMPLE`	samples from a set of units, possibly stratified by factors (procedure)
`SVSAMPLE`	constructs stratified random samples (procedure)
`APPEND`	appends values of a list of vectors of compatible types (4.4.4)
`STACK`	combines several data sets by "stacking" the corresponding vectors (4.4.5)
`UNSTACK`	splits vectors into individual vectors according to levels of a factor (4.4.6)
`JOIN`	joins or merges two sets of vectors together, based on classifying keys (4.4.7)
`FUNIQUEVALUES`	redefines a variate or text so that its values are unique (procedure)
`MVFILL`	replaces missing values in a vector with the previous non-missing value (procedure)
`RESHAPE`	reshapes a data set with classifying factors for rows and columns, into a reorganized data set with new identifying factors (procedure)
`VEQUATE`	equates values across a set of data structures (procedure)

The `APPEND` and `SUBSET` procedures are used by the Append and Subset menus in Genstat *for Windows*. The spreadsheet facilities of Genstat *for Windows* also provide several convenient menus for data manipulation, accessed by clicking Spread on the menu bar and then selecting Manipulate. For example, you can stack and unstack columns, transpose the sheet, append new data onto the ends of the columns, and so on. These facilities will generally be easier to use than the corresponding Genstat commands. Details can be found in the Spreadsheet Help file (click Help on the menu bar, and then select Spreadsheet).

There are several commands for calculations and manipulation that form variates.

`INTERPOLATE`	calculates variates of interpolated values (4.5.1)
`MONOTONIC`	fits an increasing monotonic regression (4.5.2)
`TX2VARIATE`	converts a text structure into a variate (4.5.3)
`ORTHPOLYNOMIAL`	calculates orthogonal polynomials (procedure)
`SPLINE`	calculates a set of basis functions for M-, B- or I-splines

	(procedure)
`LSPLINE`	calculates design matrices to fit a natural polynomial or trignometric L-spline as a linear mixed model (procedure)
`NCSPLINE`	calculates natural cubic spline basis functions, for use e.g. in `REML` (procedure)
`PENSPLINE`	calculates design matrices to fit a penalized spline as a linear mixed model (procedure)
`PSPLINE`	calculates design matrices to fit a P-spline as a linear mixed model (procedure)
`RADIALSPLINE`	calculates design matrices to fit a radial-spline surface as a linear mixed model (procedure)
`TENSORSPLINE`	calculates design matrices to fit a tensor-spline surface as a linear mixed model (procedure)
`VINTERPOLATE`	performs linear and inverse linear interpolation between variates (procedure)

Other commands are designed specifically for factors.

`GROUPS`	forms a factor from a variate or text, together with the set of distinct values that occur (4.6.1)
`FACAMEND`	permutes the levels and labels of a factor (procedure)
`FACDIVIDE`	represents a factor by factorial combinations of a set of factors (procedure)
`FACGETLABELS`	obtains the labels for a factor if it has been defined with labels, or constructs labels from its levels otherwise (procedure)
`FACLEVSTANDARDIZE`	redefines a list of factors so that they have the same levels or labels (procedure)
`FACMERGE`	merges levels of factors (procedure)
`FACPRODUCT`	forms a factor with a level for every combination of other factors (procedure)
`FACSORT`	sorts the levels of a factor according to an index vector (procedure)
`FDISTINCTFACTORS`	checks sets of factors to remove any that define duplicate classifications (procedure)
`FWITHINTERMS`	forms factors to define terms representing the effects of one factor within another factor (procedure)
`QFACTOR`	allows the user to decide whether to convert texts or variates to factors (procedure)

Text handling facilities include the ability to omit complete lines, or to append one text onto the end of another, using the non-specialist commands `EQUATE` and `APPEND` already mentioned. There are also several specialized commands for constructing, editing or searching texts.

`CONCATENATE`	concatenates together lines of text vectors (4.7.1)
`TXBREAK`	breaks a text structure into individual words (4.7.6)
`TXCONSTRUCT`	forms a text structure by appending or concatenating values of scalars, variates, texts, factors or pointers; allows the case of letters to be changed or values to truncated and reversed (4.7.2)
`TXFIND`	finds a subtext within a text structure (4.7.4)
`TXPAD`	pads strings of a text structure with extra characters so that their lengths are equal
`TXPOSITION`	locates strings within the lines of a text structure (4.7.3)

`TXREPLACE`	replaces strings within a text structure (4.7.5)
`TXSPLIT`	splits a text into individual texts, at positions on each line marked by separator characters (4.7.7)
`TXINTEGERCODES`	converts textual characters to and from their corresponding integer codes (4.7.8)
`TXPROGRESSION`	forms a text containing a progression of strings (4.7.9)
`EDIT`	line editor for units of text vectors (4.7.10)
`FVSTRING`	forms a string listing the identifiers of a set of data structure

Formulae can be interpreted, modified to operate on different data structures, or constructed automatically from pointers.

`FCLASSIFICATION`	forms classification sets for the terms in a formula, or breaks a formula up into separate formulae one for each term (4.8.1)
`REFORMULATE`	modifies a formula or an expression to operate on a different set of data structures (4.8.4)
`SET2FORMULA`	forms a model formula with the structures contained in a pointer (4.8.3)

You can find out which data structures are used in an expression.

`FARGUMENTS`	forms lists of data structures used as arguments in an expression (4.8.2)

Values can be assigned to dummies and pointers by the `ASSIGN` directive.

`ASSIGN`	sets values of dummies and pointers (4.9.1)

There are several commands for calculations on matrices (either as individual structures, or as elements of a compound structure such as an LRV or an SSPM).

`SVD`	calculates the singular-value decomposition of a matrix (4.10.1)
`FLRV`	calculates latent roots and vectors – that is, eigenvalues and eigenvectors (4.10.2)
`FSSPM`	calculates values for SSPM structures i.e. sums of squares and products, means, etc. (4.10.3)
`QRD`	calculates the QR decomposition of a matrix (4.10.4)
`FCORRELATION`	forms and tests the correlation matrix for a list of variates (procedure)
`FROWCANONICALMATRIX`	puts a matrix into row canonical, or reduced row echelon, form (procedure)
`FVCOVARIANCE`	forms the variance-covariance matrix for a list of variates (procedure)
`LINDEPENDENCE`	finds the linear relations associated with matrix singularities (procedure)
`MPOWER`	forms integer powers of a square matrix (procedure)
`PARTIALCORRELATIONS`	calculates a matrix of partial correlations between a set of variates (procedure)
`POSSEMIDEFINITE`	calculates a positive semi-definite approximation of a non-positive semi-definite symmetric matrix (procedure)
`STANDARDIZE`	standardizes columns of a matrix, or a set of variates, to have mean 0 and variance 1 (procedure)
`VMATRIX`	copies values and row/column labels from a matrix to variates or texts (procedure)

Tables can be formed containing summaries of values in variates: totals, minimum and maximum values, quantiles, numbers of missing and non-missing values, means and variances. The table manipulation facilities include the ability to add various types of marginal summaries to tables, and to combine "slices" of tables (and also of matrices or variates), calculation of tables of percentages, identification of outliers, and formation of a data matrix (variate and factors) from a table. You can also tabulate results from stratified surveys and surveys involving multiple-response factors.

`TABULATE`	forms tables of summaries of the values of a variate (4.11.1)
`MARGIN`	calculates or deletes margins of tables (4.11.2)
`COMBINE`	combines or omits "slices" of tables, matrices or variates (4.11.4)
`MEDIANTETRAD`	gives robust identification of multiple outliers in 2-way tables (procedure)
`PERCENT`	expresses the body of a table as percentages of one of its margins (4.11.3)
`T%CONTROL`	expresses tables as percentages of control cells (4.11.3)
`TABINSERT`	inserts the contents of a sub-table into a table (4.11.5)
`TABMODE`	forms summary tables of modes (procedure)
`TABSORT`	sorts tables so their margins are in ascending or descending order, as in a Pareto chart (4.11.6)
`TCOMBINE`	combines several tables into a single table (procedure)
`DTABLE`	plots tables (4.11.7)
`VTABLE`	forms a variate and a set of classifying factors from a table (procedure)
`FMFACTORS`	forms a pointer of factors representing a multiple-response (4.11.8)
`FFREERESPONSEFACTOR`	forms multiple-response factors from free-response data (4.11.9)
`MTABULATE`	forms tables classified by multiple-response factors (4.11.10)
`SVBOOT`	bootstraps data from random surveys (procedure)
`SVCALIBRATE`	performs generalized calibration of survey data (procedure)
`SVGLM`	fits generalized linear models to survey data (procedure)
`SVHOTDECK`	performs hot-deck and model-based imputation for survey data (procedure)
`SVMERGE`	merges strata prior to survey analysis (procedure)
`SVREWEIGHT`	modifies survey weights adjusting to ensure that their overall sum weights remains unchanged (procedure)
`SVSTRATIFIED`	analyses stratified random surveys by expansion or ratio raising (procedure)
`SVTABULATE`	tabulates data from random surveys, including multistage surveys and surveys with unequal probabilities of selection (procedure)
`SVWEIGHT`	forms survey weights (procedure)

Directives are available for adding and removing branches of trees. There are also procedures for constructing, displaying and pruning trees, which provide basic utilities for Genstat's tree-based analysis including classification trees (2:6.20), identification keys (2:6.21) and regression trees (2:3.9).

`BCUT`	cuts a tree at a defined node, discarding nodes and information below it (4.12.4)
`BJOIN`	extends a tree by joining another tree to a terminal node (4.12.5)
`BGROW`	adds new branches to a node of a tree (4.12.3)
`BCONSTRUCT`	constructs a tree (4.12.6)
`BASSESS`	assesses potential splits for regression and classification trees (4.12.7)
`BGRAPH`	plots a tree (4.12.2)
`BPRINT`	displays a tree (4.12.1)
`BPRUNE`	prunes a tree using minimal cost complexity (4.12.8)
`BIDENTIFY`	identifies specimens using a tree (4.12.9)

There are also various specialist mathematical facilities

`NAG`	calls an algorithm from the NAG Library (4.13)
`FHADAMARDMATRIX`	forms Hadamard matrices (procedure)
`FPARETOSET`	forms the Pareto optimal set of non-dominated groups (procedure)
`FPROJECTIONMATRIX`	forms a projection matrix for a set of model terms (procedure)
`FRTPRODUCTDESIGNMATRIX`	forms summation, or relationship, matrices for model terms (procedure)
`GALOIS`	forms addition and multiplication tables for a Galois finite field (procedure)
`BPCONVERT`	converts bit patterns between integers, pointers of set bits and textual descriptions (procedure)
`NCONVERT`	converts integers between base 10 and other bases (procedure)
`PERMUTE`	forms all possible permutations of the integers 1...*n* (procedure)
`PRIMEPOWER`	decomposes a positive integer into its constituent prime powers (procedure)

4.1 Numerical calculations

The main directive for calculations in Genstat is called `CALCULATE`, and this is described in the first part of this section. The calculation to be done is defined by a Genstat expression. The formal rules for these are given in 1.6.2, but below we give examples that explain in more detail exactly how they are used. Expressions also occur in `RESTRICT` (4.4.1), the directives for program control (5.2), `FITNONLINEAR` (2:3.8), or anywhere that Genstat is taking input from an identifier or list of identifiers (1.5.3). So this section is relevant also to many other areas of Genstat.

Section 4.1.1 contains general information about `CALCULATE`, describing its options and illustrating the operators that can occur in expressions. Information about calculations with particular data structures is given in 4.1.2 (scalars, factors, variates and texts), 4.1.3 (matrices) and 4.1.4 (tables). The rules for implicit declarations in `CALCULATE` are given in 4.1.5. Section 4.1.6 describes how to define subsets of vectors or matrices using qualified identifiers. The functions that can be used in expressions are described in 4.2.

4.1.1 The CALCULATE directive

CALCULATE directive

Calculates numerical values for data structures.

Options

`PRINT` = *string token*	Printed output required (`summary`); default `*` i.e. no printing
`ZDZ` = *string token*	Value to be given to zero divided by zero (`missing`, `zero`); default `miss`
`TOLERANCE` = *scalar*	If the scalar is non missing, this defines the smallest non-zero number; otherwise it accesses the default value, which is defined automatically for the computer concerned
`SEED` = *scalar*	Seed to use for any random number generation during the calculation; default 0
`INDEX` = *scalar*	If the calculation has a list of structures before the assignment operator (=), the scalar indicates the position within the list of the structure currently being evaluated
`RESTRICTEDUNITS` = *variate*	Defines a "restriction" on the vectors in the expression; if this is set the calculations on those vectors will take place only on the units listed in the variate (and any restrictions of their own will be ignored)

Parameter

expression	Expression defining the calculations to be performed

The parameter of `CALCULATE` is unnamed, and is an expression. An expression (1.6.2) consists of identifier lists, operators and functions. However, for an expression to be valid in `CALCULATE`, it must include the assignment operator (=). For example

```
CALCULATE 5,6
```

will fail, with an error message, even though the list `5,6` is an expression.

The simplest form of expression in `CALCULATE` merely assigns values from one structure to another of the same type. For example:

```
VARIATE [VALUES=1...4] V1
CALCULATE V2 = V1
```

The values of variate `V1` are copied into the structure `V2`; since `V2` has not been declared previously Genstat defines it implicitly, here as a variate. The rules for implicit declarations in `CALCULATE` are described in 4.1.5, but you may prefer to declare everything explicitly until you are confident in the use of `CALCULATE`.

A complete list of the operators available for expressions is given in 1.4.6. Most of the operators in this list act element-by-element on the values of data structures of the same type, the exceptions being the compound operator `*+` (matrix multiplication) and the four relational operators: `.IS.`, `.ISNT.`, `.IN.` and `.NI.` The assignment operator (=) has been demonstrated above; the next example shows the arithmetic operators `+`, `-`, `*`, `/` and `**` operating element-by-element on variates, `X` and `Y`:

Example 4.1.1a

```
   2  VARIATE [VALUES=10,12,14,16,*,20] X
   3  VARIATE [VALUES=4,3,2,1,0,-1] Y
```

```
CALCULATE Vadd = X + Y
& Vsub = X - Y
& Vmult = X * Y
& Vdiv = X / Y
& Vexp = X ** Y
PRINT X,Y,Vadd,Vsub,Vmult,Vdiv,Vexp; FIELDWIDTH=9; DECIMALS=2

        X        Y     Vadd     Vsub    Vmult     Vdiv     Vexp
    10.00     4.00    14.00     6.00    40.00     2.50 10000.00
    12.00     3.00    15.00     9.00    36.00     4.00  1728.00
    14.00     2.00    16.00    12.00    28.00     7.00   196.00
    16.00     1.00    17.00    15.00    16.00    16.00    16.00
        *     0.00        *        *        *        *        *
    20.00    -1.00    19.00    21.00   -20.00   -20.00     0.05
```

A missing value in either or both of the variates produces a missing value in the resulting variate.

You can use the operator minus (`-`) in two ways: either as a *dyadic* minus, to subtract one operand from another, as shown above; or as a *monadic* minus, to change the sign of a single operand. Genstat gives the monadic minus high precedence, which means that when it appears in an expression, it is one of the first operations to be done. Thus you need to be careful when using monadic minus to change the sign of the result of an expression. In particular, these two `CALCULATE` statements will give the same values to both `Vb` and Vc:

```
CALCULATE Vb = Va**2
CALCULATE Vc = -Va**2
```

This is because the operator `-` appears as a monadic minus, and so the signs of the values of `Va` are changed *before* being squared; to obtain the negative of the square of `Va` you need

```
CALCULATE Vc = -(Va**2)
```

In logical and relational expressions, Genstat uses the value 0 to represent false, and the value 1 to represent true. In fact any non-zero non-missing value is taken to represent a true value.

For numerical structures, Genstat has the relational operators `.EQ.`, `.NE.`, `.LT.`, `.LE.`, `.GT.` and `.GE.` with their symbolic equivalents. Note that the symbolic equivalent for `.EQ.` is the compound operator ==. Genstat requires the two (adjacent) equals signs to distinguish this from the assignment operator, which would generate rather different results! For text structures, the appropriate relational operators are `.EQS.` and `.NES.` The two texts must have the same number of units (or lines).

Example 4.1.1b

```
VARIATE [VALUES=1,2,3,4,5,*,*,1] X
& [VALUES=5,4,3,2,1,1,*,*] Y
TEXT [VALUES=a,b,c,d,e,'','',a] Tx
& [VALUES=a,x,c,d,y,a,'',''] Ty
CALCULATE Veq = X.EQ.Y
& Vne = X.NE.Y
& Vlt = X.LT.Y
& Vle = X.LE.Y
& Vgt = X.GT.Y
& Vge = X.GE.Y
& Veqs = Tx.EQS.Ty
& Vnes = Tx.NES.Ty
PRINT X,Y,Veq,Vne,Vlt,Vle,Vgt,Vge,Tx,Ty,Veqs,Vnes; \
  FIELDWIDTH=5; DECIMALS=0

    X    Y  Veq  Vne  Vlt  Vle  Vgt  Vge   Tx   Ty Veqs Vnes
    1    5    0    1    1    1    0    0    a    a    1    0
    2    4    0    1    1    1    0    0    b    x    0    1
    3    3    1    0    0    1    0    1    c    c    1    0
    4    2    0    1    0    0    1    1    d    d    1    0
    5    1    0    1    0    0    1    1    e    y    0    1
    *    1    0    1    *    *    *    *         a    0    1
    *    *    1    0    *    *    *    *              1    0
```

```
   1    *    0    1    *    *    *    *    a         0    1
```

With most of the relational operators, a missing value in either operand, or in both, gives a missing result. The exceptions are `.EQ.` and `.NE.`, `.EQS.` and `.NES.` When both operands are missing, `.EQ.` gives a true result and `.NE.` gives a false result. The same is true with `.EQS.` and `.NES.` when they encounter missing values (or null strings) in texts.

The relational operators `.IS.` and `.ISNT.` test whether or not a dummy points to a particular identifier. For example, to store in `Sca` the result of a test to check whether dummy `D` points to `Va`, you would put

```
CALCULATE Sca = D.IS.Va
```

while to test that `D` does not point to `Vb`, you would put

```
CALCULATE Sca = D.ISNT.Vb
```

The final pair of relational operators, `.IN.` and `.NI.`, represent *inclusion* and *non-inclusion*.

These two operators differ from the other relational operators in that each value in the structure on the left-hand side is compared in turn with every value in the structure on the right-hand side.

For `.IN.`, the result is true if the value on the left-hand side is included in the set of values in the right-hand structure; otherwise the result is false. The `.NI.` operator is the opposite of `.IN.`

The length of the result is taken from the length of the left-hand structure, since it is the values of the left-hand structure that are being tested. For a very simple example, suppose that the variate `X` contains the values 1,2,1,1,3,5,1,2,1,4, and that the variate `Evens` contains 0,2,4,6,8. The statement

```
CALCULATE S = X.IN.Evens
```

will store in `S` an indication of whether each element of `X` is odd or even: that is, `S` will be given the values 0,1,0,0,0,0,0,1,0,1.

When there is a factor on the left-hand side of `.IN.` or `.NI.` and a variate on the right-hand side, Genstat checks the levels of the factor against the values in the variate. Alternatively, if the factor has a labels vector, you can specify a text against which Genstat will then compare the labels. In the next example, the variate `Large` records which elements of the factor `Size` have values that lie in the set {4.8, 6}, and the variate `NotAB` records which elements of the factor `Type` have values that lie outside the set {`A`, `B`}:

Example 4.1.1c

```
   2  FACTOR [LEVELS=!(1.2,2.4,3.6,4.8,6)] Size; \
   3    VALUES=!(1.2,4.8,6,2.4,3.6,2.4,1.2,6)
   4  FACTOR [LABELS=!T(A,B,C,D)] Type; VALUES=!T(2(A,B,C,D))
   5  CALCULATE Large = Size .IN. !(4.8,6)
   6  & NotAB = Type .NI. !T(A,B)
   7  PRINT Size,Large,Type,NotAB; FIELDWIDTH=6; DECIMALS=1,0,0,0

 Size Large  Type NotAB
  1.2     0     A     0
  4.8     1     A     0
  6.0     1     B     0
  2.4     0     B     0
  3.6     0     C     1
  2.4     0     C     1
  1.2     0     D     1
  6.0     1     D     1
```

You can use the logical operators `.AND.`, `.NOT.`, `.OR.` and `.EOR.` to combine the results of the relational operators, and form a single logical result: `.NOT.` reverses true and false results; `.OR.` gives a true result only if one or both operands are true; `.AND.` gives a true result if both operands are true; and `.EOR.` gives a true result if one of the operands is true, but a false result if both are true or both false. Example 4.1.1d shows the results of the four logical operators. Notice that a missing value in either operand gives a missing value in the result.

Example 4.1.1d

```
VARIATE [VALUES=3(0,1,2),1,*] X
& [VALUES=(0,1,2)3,*,*] Y
CALCULATE Vnot = .NOT. Y
& Vor = X .OR. Y
& Vand = X .AND. Y
& Veor = X .EOR. Y
PRINT X,Y,Vnot,Vor,Vand,Veor; FIELDWIDTH=5; DECIMALS=0

    X    Y Vnot  Vor Vand Veor
    0    0    1    0    0    0
    0    1    0    1    0    1
    0    2    0    1    0    1
    1    0    1    1    0    1
    1    1    0    1    1    0
    1    2    0    1    1    0
    2    0    1    1    0    1
    2    1    0    1    1    0
    2    2    0    1    1    0
    1    *    *    1    *    *
    *    *    *    *    *    *
```

If the expression contains lists, Genstat does several calculations. For example,

```
CALCULATE A,B,C = X,Y,Z + 1,2,3
```

is equivalent to the three `CALCULATE` statements:

```
CALCULATE A = X + 1
CALCULATE B = Y + 2
CALCULATE C = Z + 3
```

Genstat takes the items in the lists in parallel, and recycles any lists that are shorter than the list of primary arguments. In `CALCULATE`, the primary arguments are the identifiers on the left-hand side of the assignment operator (=). In the above example, each list had three identifiers, and so no recycling was done; but in the statement

```
CALCULATE A,B,C = X,Y + 1,2,3
```

the second list is of length only two, and so is recycled to give the calculations:

```
CALCULATE A = X + 1
CALCULATE B = Y + 2
CALCULATE C = X + 3
```

The `INDEX` option allows you to define a scalar which will store the number of the calculation that is taking place. So, we could do the calculations above by putting

```
CALCULATE [INDEX=Ncalc] A,B,C = X,Y + Ncalc
```

The scalar `Ncalc` takes the value 1 while `A` is calculated (i.e. during the first calculation), then 2 while `B` is calculated, and finally 3 while `C` is calculated.

If the longest list is not on the left-hand side of the assignment operator, `CALCULATE` gives a fault diagnostic.

We must stress that Genstat operates on lists of *data structures* in its calculations and not on lists of *expressions*; if you want to specify several expressions, you must separate them by semicolons and not commas. As an example, suppose that you have two variates `X` and `Y`, where `X` is to be multiplied by `10`, and `Y` is to be divided by `180`. Naively, you might write the statement:

```
CALCULATE X,Y = X*10,Y/180
```

This statement is syntactically correct, but it does not do what you want: in fact it corresponds to the pair of statements

```
CALCULATE X = X*10/180
CALCULATE Y = X*Y/180
```

which is quite different from the intention. Genstat interprets the elements of the list on the right-hand side as `10` and `Y`, and not as "`X*10`" and "`Y/180`". If you really want to combine these two operations together in a single expression, you need to put

```
CALCULATE X,Y = X,Y * 10,1 / 1,180
```

Then the three lists on the right-hand side are taken in parallel: firstly `X`, `10` and `1`, and then `Y`, `1` and `180`. If you want to execute more than one expression in a `CALCULATE` statement, you must separate each one from the next by a semicolon: for example

```
CALCULATE X = X*10; Y = Y/180
```

`CALCULATE` has three further options: `PRINT`, `ZDZ` and `TOLERANCE`. If you set the `PRINT` option to `summary`, Genstat will print some summary information every time that values are assigned to a structure. The information has the same form as in the `READ` directive (3.1.2): identifier, minimum value, mean value, maximum value, number of values, number of missing values, and whether or not the set of values is skew. In Example 4.1.1e two assignments are made, and summaries are printed for the variates `B` and `C`.

Example 4.1.1e

```
   2  VARIATE [VALUES=1,4,*,7,10] A
   3  CALCULATE [PRINT=summary] C = (B = 2*A) + 1

    Identifier   Minimum      Mean   Maximum     Values   Missing
             B     2.000     11.00     20.00          5         1
             C     3.000     12.00     21.00          5         1
```

If you try to use `CALCULATE` to do something invalid, such as the logarithm or the square root of a negative number, Genstat generates a warning diagnostic and inserts a missing value in the offending unit. The one exception is the division of zero by zero, which is regarded as deliberate. Genstat thus does not print a diagnostic, but uses option `ZDZ` to determine whether the result should be a missing value (`ZDZ=missing`) or zero (`ZDZ=zero`); the default is `missing`. In this example, the variate `%dm` is formed with zeroes in the positions where `Fresh_wt` and `Dry_wt` both have zeroes.

Example 4.1.1f

```
   2  VARIATE [VALUES=15.74,88.61,48.70,0,49.37] Fresh_wt
   3  & [VALUES=3.21,11.3,7.83,0,7.23] Dry_wt
   4  CALCULATE [ZDZ=zero] %dm = 100*Dry_wt / Fresh_wt
   5  PRINT Dry_wt,Fresh_wt,%dm; FIELDWIDTH=9; DECIMALS=2

  Dry_wt Fresh_wt      %dm
    3.21    15.74    20.39
   11.30    88.61    12.75
    7.83    48.70    16.08
    0.00     0.00     0.00
    7.23    49.37    14.64
```

Arithmetic operations with real numbers can suffer from rounding errors. Genstat uses real arithmetic for all its operations in `CALCULATE`, and so makes allowance for cases where rounding error may cause problems: in other words, very small numbers are taken to be zero. Sometimes, however, you may want to do calculations with numbers that are genuinely very small, and so the `TOLERANCE` option allows you to change the value that Genstat uses to assess the rounding-off.

4.1.2 Expressions with scalars and vectors

Example 4.1.2a shows a calculation involving scalars and variates. Several scalars are used to transform the variate `Mpg` (miles per gallon) into its metric counterpart `Lp100k` (litres per 100 kilometres). The scalar values are applied to every unit of the variate. You will see that the zero value in `Mpg` causes Genstat to print the warning "`Attempt to divide by zero`"; a missing value is placed in the corresponding position in `Lp100k`. This warning is printed only once per operation; so subsequent zero values in `Mpg` do not trigger it again.

Example 4.1.2a

```
   2  VARIATE [VALUE=0,10,20,30,32...40,0,50] Mpg
   3  VARIATE Lp100km
   4  SCALAR Lpt,Cmin,Ydm,Inyd,Mkm; VALUE=0.568,2.54,1760,36,1000
   5  CALCULATE Lp100km = 8 * Lpt * 100 * Mkm * 100 / \
   6    ( Mpg * Ydm * Inyd * Cmin )

******** Warning 1, Code CA 18, statement 1 on Line 6

Command: CALCULATE Lp100km = 8 * Lpt * 100 * Mkm * 100 /  ( Mpg * Ydm
* Inyd * Cmin )
Attempt to divide by zero.
Attempt to divide by zero occurs at unit 1

   7  PRINT Lp100km,Mpg; FIELDWIDTH=8; DECIMALS=2

 Lp100km     Mpg
       *    0.00
   28.24   10.00
   14.12   20.00
    9.41   30.00
    8.82   32.00
    8.30   34.00
    7.84   36.00
    7.43   38.00
    7.06   40.00
       *    0.00
    5.65   50.00
```

If you use `CALCULATE` with variates that have been "restricted" using the `RESTRICT` directive (4.4.1), Genstat applies the same restriction to all the variates involved in each calculation. Thus if more than one of these variates is restricted, they must all be restricted in the same way. In Example 4.1.2b, the variate Fresh_wt is restricted to those of its values that correspond to the first level of the factor `Block`. Only these units are involved in the calculation; the other units are left unchanged. Here the variate `%dm` had no values, and so the units in block 2 are given missing values.

Example 4.1.2b

```
   2  VARIATE [VALUES=15.74,88.61,48.70,49.37,18.96,12.13,23.38,48.16]\
   3    Fresh_wt
   4  & [VALUES=3.21,11.3,7.83,7.23,3.55,2.6,4.0,6.43] Dry_wt
   5  VARIATE [NVALUES=8] %dm
   6  FACTOR [LEVELS=2; VALUES=4(1,2)] Block
   7  RESTRICT Fresh_wt; CONDITION=Block.EQ.1
   8  CALCULATE %dm = Dry_wt*100/Fresh_wt
   9  PRINT %dm
```

```
      %dm
    20.39
    12.75
    16.08
    14.64
        *
        *
        *
        *
```

The RESTRICTEDUNITS option provides an alternative way of specifying a restriction. Its setting is a variate containing a list of the units numbers on which you want the calculation to be done (the other units are then ignored). This works in the same way as if you had applied a restriction on one of these vectors explicitly. However, if RESTRICTEDUNITS is set, restrictions on the vectors themselves are ignored. By default, when RESTRICTEDUNITS is unset, CALCULATE will look for restrictions in the vectors, as usual. However, you can set RESTRICTEDUNITS=* to make the calculation work on all the units, regardless of whether any of the vectors is restricted.

Note, though, that restrictions on a variate within a scalar function (for example MEAN; 4.2.2), or within the RESTRICTION function (4.2.8), operate independently from the main calculation outside. Also, restrictions are ignored if the main calculation contains qualified identifiers or the ELEMENTS function (4.2.8).

When Genstat implicitly declares a structure during a CALCULATE operation, it also by default sets its attributes to match those of the structures in the calculation: for details, see 4.1.5. We now do the same calculation, but leave Genstat to declare the structure %dry_m as a variate. In particular, the length of %dry_m will be the same as that of Fresh_wt, and %dry_m will become restricted in the same way as Fresh_wt. Thus, the PRINT statement shows only the values of %dry_m corresponding to block 1; in fact, all the other values of %dry_m are missing. Genstat would also have carried the restriction across if we had declared %dry_m as a variate but had left the CALCULATE statement to set its number of values.

Example 4.1.2c

```
  10  CALCULATE %dry_m = Dry_wt*100/Fresh_wt
  11  PRINT %dry_m

      %dry_m
       20.39
       12.75
       16.08
       14.64
```

If you put a factor in a calculation, Genstat will use its levels, as shown when variate V takes its values from the factor F in line 3 of Example 4.1.2d. The function NEWLEVELS (4.2.1) allows you to specify an alternative levels variate to be used instead in the calculation. Line 4 of Example 4.1.3d uses the values 3.5 and 6.4, instead of the values 2 and 4 in the levels variate of the factor F, when forming the values of the variate Vn.

Example 4.1.2d

```
   2  FACTOR [LEVELS=!(2,4)] F; VALUES=!((2,4)4)
   3  CALCULATE V = F
   4  & Vn = NEWLEVELS(F; !(3.5,6.4))
   5  PRINT V,Vn

          V          Vn
      2.000       3.500
      4.000       6.400
      2.000       3.500
      4.000       6.400
      2.000       3.500
      4.000       6.400
      2.000       3.500
      4.000       6.400
```

If the factor is on the left-hand side of the equals sign, Genstat checks that each of the results of the calculation is an acceptable level. This allows you to define the values of a factor from a variate, or from another factor. However you must already have declared the factor, with its levels and labels vectors; factors cannot be declared implicitly. Example 4.1.2e first sets the values of the factor `Rate` from the variate `Setting`; it then uses the `NEWLEVELS` function to form the values of the factor `Amount`, whose first level corresponds to levels 1 and 2 of the factor `Rate` and whose second level corresponds to levels 3 and 4.

Example 4.1.2e

```
   2  VARIATE [VALUES=1,3,2,1,4,3,1,2] Setting
   3  FACTOR [LEVELS=!(1.25,2.5,3.75,5)] Rate
   4  CALCULATE Rate = Setting*1.25
   5  FACTOR [LABELS=!T(lower,higher)] Amount
   6  CALCULATE Amount = NEWLEVEL(Rate; !(1,1,2,2))
   7  PRINT Setting,Rate,Amount; FIELDWIDTH=8; DECIMALS=2

 Setting     Rate  Amount
    1.00     1.25   lower
    3.00     3.75  higher
    2.00     2.50   lower
    1.00     1.25   lower
    4.00     5.00  higher
    3.00     3.75  higher
    1.00     1.25   lower
    2.00     2.50   lower
```

Text structures are allowed only with the relational operators `.EQS.`, `.NES.`, `.IN.` and `.NI.` described in 4.1.1, or in the functions `CHARACTERS`, `GETFIRST`, `GETLAST`, `GETPOSITION`, `NOBSERVATIONS`, `NMV`, `NVALUES` and `POSITION`. The result of any expression is a number, so you cannot create a text with `CALCULATE`, even if the structures on which the operations are being done are texts.

4.1.3 Expressions with matrices

All the arithmetic, relational and logical operators that we have now seen in use with variates can also be used with rectangular matrices, symmetric matrices and diagonal matrices. The basic rule when using these with different types of matrix is that their dimensions must conform. This means that, for each pair of matrices, row dimension must match row dimension, and column dimension must match column dimension. Consider the matrices `Mx`, `My` and `Mz`, and the symmetric matrix `Smz` declared here:

```
     MATRIX [ROWS=3; COLUMNS=4] Mz,My
     MATRIX [ROWS=3; COLUMNS=3] Mx
     SYMMETRICMATRIX [ROWS=3] Smz
```

The dimensions of `Mz` and `My` conform; but the dimensions of `Mx` and `Mz` do not, since `Mx` and `Mz` have different numbers of columns, three and four respectively. Similarly the dimensions of the symmetric matrix `Smz` and the matrix `Mx` conform; but the dimensions of `Smz` and `Mz` do not.

For simplicity, our examples mostly involve addition; but remember that you can replace the operator + with any of the other arithmetic, logical or relational operators. Matrix multiplication is described towards the end of this subsection.

In Example 4.1.3a, two rectangular matrices, `Ma` and `Mb` (each with four rows and three columns) are added together to form `Mc`. Note that Genstat operates in turn on each element of these two matrices, and that the new structure `Mc` is a matrix also with four rows and three columns.

Example 4.1.3a

```
MATRIX [ROWS=4; COLUMNS=3] Ma,Mb; \
  VALUES=!(1...12),!(5,4,6,12,10,11,7,9,8,3,1,2)
& Mc
CALCULATE Mc = Ma + Mb
PRINT Mc

                Mc
                 1           2           3

        1     6.00        6.00        9.00
        2    16.00       15.00       17.00
        3    14.00       17.00       17.00
        4    13.00       12.00       14.00
```

When you do calculations with two diagonal matrices, each one must have the same number of rows. Similarly, with symmetric matrices, the row dimensions must match. When you add, subtract, multiply, divide or exponentiate a symmetric matrix, only those elements that are stored by Genstat are operated on. Here the two symmetric matrices `Sma` and `Smb` are added together to form another symmetric matrix `Smc`; this is done element by element.

Example 4.1.3b

```
SYMMETRICMATRIX [ROWS=4] Sma,Smb; \
  VALUES=!(1...10),!(7,8,4,9,5,2,10,6,3,1)
& Smc
CALCULATE Smc = Sma + Smb
PRINT Smc

         Smc

 1       8.00
 2      10.00        7.00
 3      13.00       10.00        8.00
 4      17.00       14.00       12.00       11.00
            1           2           3           4
```

If you use a symmetric matrix in a calculation together with a matrix, it will be extended to include the values above the diagonal, before the calculation is done. Similarly, diagonal matrices are extended for calculations with matrices or symmetric matrices. Example 4.1.3c adds the diagonal matrix `Da` to the symmetric matrix `Sma` and puts the results in the matrix `Md`.

Example 4.1.3c

```
DIAGONALMATRIX [ROWS=4; VALUES=3,2,4,1] Da
MATRIX [ROWS=4; COLUMNS=4] Md
CALCULATE Md = Sma + Da
PRINT Md

                Md
                 1           2           3           4

        1    4.000       2.000       4.000       7.000
        2    2.000       5.000       5.000       8.000
        3    4.000       5.000      10.000       9.000
```

```
          4       7.000       8.000       9.000      11.000
```

You can also use variates together with matrices, provided their dimensions conform. Genstat treats variates as column matrices: that is, with *n* rows and one column. Example 4.1.3d adds the variate `Va` to the four-by-one matrix `Me`.

Example 4.1.3d

```
VARIATE [NVALUE=4; VALUES=4,2,1,3] Va
MATRIX [ROWS=4; COLUMNS=1; VALUES=10,4,7,2] Me
CALCULATE Me = Me + Va
PRINT Me

                Me
                 1

          1      14.000
          2       6.000
          3       8.000
          4       5.000
```

You can use a scalar with any of the matrix structures; the scalar is applied to every element of the matrix, in exactly the same way as when scalars and variates occur together in a calculation (4.1.2). Here the scalar `Sca` is added to every element of the symmetric matrix `Sma`.

Example 4.1.3e

```
SCALAR Sca; VALUE=3
CALCULATE Sma = Sma + Sca
PRINT Sma

            Sma

  1       4.000
  2       5.000       6.000
  3       7.000       8.000       9.000
  4      10.000      11.000      12.000      13.000
              1           2           3           4
```

The multiplication operator (`*`) means element-by-element multiplication for the two matrices, not matrix multiplication.

Example 4.1.3f

```
MATRIX [ROWS=4; COLUMNS=3] Mf
CALCULATE Mf = Ma * Mb
PRINT Mf

                Mf
                 1           2           3

          1       5.00        8.00       18.00
          2      48.00       50.00       66.00
          3      49.00       72.00       72.00
          4      30.00       11.00       24.00
```

For matrix multiplication you can use the compound operator `*+` or the function `PRODUCT` (4.2.4). The column dimension of the first matrix must then match the row dimension of the second. In Example 4.1.3g, the four-by-four matrix `Mh` is formed from the matrix product of `Ma` with Mg, a matrix with three rows and four columns.

Example 4.1.3g

```
MATRIX [ROWS=3; COLUMNS=4; VALUES=1,4,7,10,2,5,8,11,3,6,9,12] Mg
MATRIX [ROWS=4; COLUMNS=4] Mh
CALCULATE Mh = Ma *+ Mg
PRINT Mh

                  Mh
                   1           2           3           4

           1    14.0        32.0        50.0        68.0
           2    32.0        77.0       122.0       167.0
           3    50.0       122.0       194.0       266.0
           4    68.0       167.0       266.0       365.0
```

To summarize then, `*+` is used in Genstat for matrix multiplication while `*` allows the corresponding elements of two matrices to be multiplied together.

The rules for implicit declarations when combining matrices are in 4.1.5. The rules for qualified identifiers of matrices are in 4.1.6. Genstat provides several special matrix functions, including the `INVERSE` and `GINVERSE` functions, which can be included in `CALCULATE` statements; for details see 4.2.4. There are also several specialized functions for manipulating RGB images stored in matrices; see 4.2.13 and 6.5.1.

4.1.4 Expressions with tables

You can use tables in expressions in much the same way as you would any other numerical structure. Arithmetic, relational and logical operators act element-by-element, as do the general functions (4.2.1).

Tables in expressions must be either all without margins or all with margins. If you try to mix tables with and without margins, Genstat will report an error.

Calculations with tables are very straightforward when they have the same factors in their classifying sets. In Example 4.1.4a two tables are added together:

Example 4.1.4a

```
FACTOR [LEVELS=2; LABELS=!T(Woburn,Rothamsted)] Soil
& [LEVELS=2; LABELS=!T(low,medium)] Acidity
TABLE [CLASSIFICATION=Soil,Acidity] Ta,Tb; \
  VALUES=!(6.91,4.98,4.86,*),!(6.38,4.68,6.49,*)
& Tc
CALCULATE [PRINT=summary] Tc = Ta + Tb

 Identifier   Minimum      Mean   Maximum    Values   Missing
         Tc     9.660     11.43     13.29         4         1

PRINT Tc

                  Tc
   Acidity       low      medium
      Soil
    Woburn     13.29        9.66
Rothamsted     11.35           *
```

When tables have different classifying sets, there are two cases to consider. We illustrate them with the assignment operator, but the rules apply to any operation. The first case is when the table on the left-hand side has a factor in its classifying set that is not in the classifying set of the table on the right-hand side. In this case, the right-hand table is expanded to include that factor, by duplicating its values across the levels of the factor and any margin. Thus, in Example 4.1.4b, the values of the table Tb are repeated over the levels of the factor Block, which is the factor additional in the table Td. In other words the table Tb has been extended to include the factor Block: perhaps the easiest way of thinking about what happens is that each level of the extra factor contains a whole copy of the table on the right-hand side.

Example 4.1.4b

```
FACTOR [LEVELS=2] Block
TABLE [CLASSIFICATION=Soil,Acidity,Block] Td
CALCULATE Td = Tb
PRINT Td

                                     Td
                   Block              1           2
       Soil      Acidity
     Woburn          low          6.380       6.380
                  medium          4.680       4.680
 Rothamsted          low          6.490       6.490
                  medium              *           *
```

The second case is when the table on the right-hand side has a factor in its classifying set that is not in the classifying set of the table on the left-hand side. Now the values in the margin over that factor are taken for the left-hand table. If the table has no margins, they must be calculated first. By default Genstat forms marginal totals, but you can use the special table functions (4.2.5) to form other types of margin. In Example 4.1.4c, marginal totals are calculated for table Td over the factor Block, and the results are placed in the previously declared table Tc.

Example 4.1.4c

```
CALCULATE Tc = Td
PRINT Tc

                       Tc
    Acidity           low      medium
       Soil
     Woburn         12.76        9.36
 Rothamsted         12.98           *
```

The classifying set of a table has two forms – one taken from the sequence in which the factors were listed in the CLASSIFICATION option of the TABLE declaration (2.5), the other determined by the order in which the identifiers of the factors are stored within Genstat. The second of these is called the *ordered classifying* set, and is the one used by CALCULATE for all operations on tables. CALCULATE permutes the values of tables so that they correspond to the ordered classifying set.

There are two consequences. The first is that if a fault occurs while an operation on a table is being done, its values may have been permuted, and so may no longer be in the order corresponding to the classifying set specified in the CLASSIFICATION option of the TABLE declaration. However, this occurs only if there has been a fault, since CALCULATE does not permute the values permanently.

The second consequence concerns implicit declarations. When a table is declared implicitly there is no obvious order for the factors other than the order in which their identifiers are stored (which generally reflects the order in which they were defined within the job). Thus Genstat will define the classifying set to be the same as the ordered classifying. So, if the resulting table Tc had not already been declared in Example 4.1.4a, its classifying set would in this case have been Acidity, Soil. So, if you want your table printed with the factors in a particular order, you must declare the table before its values are assigned in CALCULATE, or else use the DOWN, ACROSS and WAFER options of PRINT (3.2.2).

4.1.5 Rules for implicit declarations

Undeclared structures on the left-hand side of an assignment (=) in an expression are declared automatically: this is known as an *implicit declaration*. The type of structure is chosen to be the one most appropriate to the results that have been produced. This can be described according to a few straightforward rules.

The assignment operator (=) can appear anywhere in an expression, and so you need to be aware of the order of evaluation. For example, in the CALCULATE statement

```
    CALCULATE Vc = Va*Vb
```

the result of Va*Vb is not placed directly in Vc: CALCULATE forms an intermediate structure whose values in this case are the results of Va*Vb; then the values of the intermediate structure are assigned to Vc. On assignment, the type and other relevant attributes of the resultant structure are also taken from the intermediate structure if these have not been defined previously (either implicitly or explicitly).

When structures of the same type are combined, the rule is that the intermediate structure will be of the same type; the same rule applies to tables with identical classifying sets. When structures of different types are combined, you need to know what form the intermediate structure takes.

In list below, .OP. refers to any arithmetic, logical or relational operator, except .IS., .ISNT., .IN., .NI., .EQS. and .NES. which have their own rules described earlier (4.1.1). The dimensions of operands must conform in any operation involving matrices and variates. The second column indicates the type of structure resulting from the operation and the third column lists the types of structures to which it can be assigned.

Combination	**Intermediate structure**	**Assignment**
Scalar .OP. Scalar	Scalar	*any*
Variate .OP. Scalar	Variate	Variate,Factor
Variate .OP. Variate	Variate	Variate,Factor
Factor .OP. Scalar	Variate	Variate,Factor
Factor .OP. Variate	Variate	Variate,Factor
Factor .OP. Factor	Variate	Variate,Factor
Diagonal .OP. Scalar	Diagonal	Diagonal,Symmetric
Diagonal .OP. Variate	*invalid*	-
Diagonal .OP. Factor	*invalid*	-
Diagonal .OP. Diagonal	Diagonal	Diagonal,Symmetric,Matrix
Symmetric.OP.Scalar	Symmetric	Diagonal,Symmetric,Matrix
Symmetric.OP.Variate	*invalid*	-
Symmetric.OP.Factor	*invalid*	-
Symmetric.OP.Diagonal	Symmetric	Diagonal,Symmetric,Matrix
Symmetric.OP.Symmetric	Symmetric	Diagonal,Symmetric,Matrix
Matrix .OP. Scalar	Matrix	Matrix
Matrix .OP. Variate	Matrix	Matrix,Variate
Matrix .OP. Factor	Matrix	Matrix,Variate
Matrix .OP. Diagonal	Matrix	Diagonal,Symmetric,Matrix
Matrix .OP. Symmetric	Matrix	Diagonal,Symmetric,Matrix
Matrix .OP. Matrix	Matrix	Matrix
Table .OP. Scalar	Table	Table
Table .OP. Variate	*invalid*	-
Table .OP. Factor	*invalid*	-
Table .OP. Diagonal	*invalid*	-

Table `.OP.` Symmetric	*invalid*	-
Table `.OP.` Matrix	*invalid*	-
Table `.OP.` Table	Table	Table

In the last rule, Table `.OP.` Table, the classifying set of the intermediate table is the union of the two classifying sets. For example, in

```
FACTOR [LEVELS=2] Fa,Fb,Fc
TABLE [CLASSIFICATION=Fa,Fb] Ta
TABLE [CLASSIFICATION=Fa,Fc] Tb
CALCULATE Tc = Ta+Tb
```

The resulting table, `Tc`, will have the classifying set `Fa`, `Fb` and `Fc`. As explained at the end of 4.1.4, the classifying set of a table has two forms. All tables in `CALCULATE` have their values permuted according to the ordered classifying set. On assignment, the ordered classifying set is transferred to the new table, which Genstat declares implicitly. So the classifying set and ordered classifying set are the same for tables declared implicitly.

The third column, headed "Assignment", lists the types of structure to which the values in the intermediate structure can be assigned. Genstat allows a fair amount of flexibility in this. All the intermediate structures contain numbers, and so you cannot declare factors implicitly in `CALCULATE`. However, you can assign a variate to a factor, so long as the values of the variate all occur as valid levels of the factor (4.1.2).

Most functions produce a result with the same type as their first argument, but there are some exceptions like the scalar and variate functions (see 4.2, and especially 4.2.2 and 4.2.3).

4.1.6 Rules for qualified identifiers

Qualified identifiers were introduced in 1.5.3, together with the rules for expanding them into lists. The rules for their use are similar to the rules for the arguments of the `ELEMENTS` function (4.2.8). the number of qualifiers that a structure can have is determined by its dimensionality. The dimensionality of scalars is defined to be zero, and so they cannot be qualified. Tables have varying numbers of dimensions, up to nine, and in the current of Genstat cannot be qualified. The dimensionalities of the structures that can be qualified are as follows.

1) variate, text, factor, diagonal matrix and symmetric matrix.
2) matrix and symmetric matrix.

Notice that a symmetric matrix can have a dimensionality of either one or two, and so can be qualified in two ways; these are described below.

The qualifiers can be scalars, numbers, variates, quoted strings, or texts. The set of units defined by a qualifier is built up, by taking its values one at a time. Positive numbers (or texts or strings) add units to the set, while negative numbers delete the corresponding units from the set (if already there). A missing value can be used to include all the units, and one of these will be included implicitly at the start of the qualification list if the first element of the list is negative.

When an expression contains several qualified vector structures, you define a different subset for each vector; but for the calculation to work, the number of values contributed from each vector must be the same: see lines 6 and 7 of Example 4.1.6a. Genstat then ignores any restrictions on the vectors; in fact qualified identifiers provide an alternative way of specifying subsets of vectors. Example 4.1.6a illustrates the use of qualifications with variates, texts and a factor. In each case the qualified vector is a vector with fewer values, but of the same type as the original structure: for example, `Ta$[!(1,3,5)]` is a text with three values instead of six.

Example 4.1.6a

```
 2  VARIATE [NVALUES=5] Va; VALUES=!(1...5)
 3  TEXT [NVALUES=6] Ta,Tb; VALUES=!T(a,b,c,d,e,f),!T(a,a,c,c,f,f)
 4  FACTOR [NVALUES=8; LEVELS=3] Fa; VALUES=!(1,3,2,3,1,2,3,1)
```

```
   5  VARIATE [VALUES=12(0)] Vb
   6  CALCULATE Vb$[!(3,6,10)] = Va$[!(1,2,5)] * \
   7    (Ta$[!(1,3,5)] .EQS. Tb$[!(2,4,6)]) + Fa$[!(5,7,2)]
   8  PRINT Vb; DECIMALS=0

          Vb
           0
           0
           2
           0
           0
           5
           0
           0
           0
           3
           0
           0
```

When you have a qualified diagonal matrix, the subset of values is itself a diagonal matrix. Similarly a symmetric matrix, qualified by a single list, is also a symmetric matrix. The qualifier indicates which rows and columns are to be included; see line 4 of Example 4.1.6b.

Example 4.1.6b

```
   2  SYMMETRICMATRIX [ROWS=4] Sma; VALUES=!(1...10)
   3  & [ROWS=3] Smb
   4  CALCULATE Smb = Sma$[!(1,4,2)]
   5  PRINT Sma,Smb; FIELDWIDTH=6; DECIMALS=0

       Sma

   1     1
   2     2     3
   3     4     5     6
   4     7     8     9    10
         1     2     3     4

       Smb

   1     1
   2     7    10
   3     2     8     3
         1     2     3

   6  MATRIX [ROWS=4; COLUMNS=5] Ma; VALUES=!(1...20)
   7  & [ROWS=2; COLUMNS=2] Mb
   8  CALCULATE Mb = Sma$[!(1,4);!(2,3)] + Ma$[!(1,4);!(3,4)]
   9  PRINT Ma,Mb; FIELDWIDTH=6; DECIMALS=0

                Ma
                 1     2     3     4     5

          1      1     2     3     4     5
          2      6     7     8     9    10
          3     11    12    13    14    15
          4     16    17    18    19    20

                Mb
                 1     2

          1      5     8
          2     26    28
```

Symmetric matrices can also have two qualifiers, in which case Genstat treats the result as a rectangular matrix. Rectangular matrices must have two qualifiers. In line 8 of Example 4.1.6b, the values of the rectangular matrix `Mb` are formed from the addition of the values in rows 1 and

4, and columns 2 and 3, of the symmetric matrix `Sma` to the values in rows 1 and 4, and columns 3 and 4, of the matrix `Ma`.

All the examples above show how to form vectors and matrices that have fewer values than the original: that is, the vectors and matrices take their values from subsets of the source structures. You can form also larger vectors and matrices, by using repeated values in the qualifier set. In Example 4.1.6c the matrix `Mc`, with four rows and three columns is formed from the two-by-two matrix `Mb`.

Example 4.1.6c

```
MATRIX [ROWS=2; COLUMNS=2; VALUES=5,7,6,2] Mb
MATRIX [ROWS=4; COLUMNS=3] Mc
VARIATE [NVALUES=4; VALUES=1,2,2,1] Va
& [NVALUES=3; VALUES=1,1,2] Vb
CALCULATE Mc = Mb$[Va; Vb]
PRINT Mc; FIELDWIDTH=6; DECIMALS=0

           Mc
            1     2     3

      1     5     5     7
      2     6     6     2
      3     6     6     2
      4     5     5     7
```

Instead of using variates to qualify the structures, you can use any numerical structure, and these structures can be qualified too. Genstat treats any structure used as a qualifier as a one-dimensional list of values. You can build very complicated qualifications in this way. The only limitation is that the set of values of the qualifiers must form a valid address list for the parent structure. In Example 4.1.6d, the complicated qualification reduces to assigning the value 3 to the element in row 3 and column 4 of the matrix `Ma`.

Example 4.1.6d

```
VARIATE [NVALUES=6] Va; VALUES=!(1,4,3,2,4,3)
MATRIX [ROWS=4; COLUMNS=6] Ma; VALUES=!(1...24)
CALCULATE Ma$[Va$[2]; Ma$[1; 3]] = 3
PRINT Ma; FIELDWIDTH=6; DECIMALS=0

           Ma
            1     2     3     4     5     6

      1     1     2     3     4     5     6
      2     7     8     9    10    11    12
      3    13    14    15    16    17    18
      4    19    20     3    22    23    24
```

You can use text to qualify structures, since it can label the rows and columns of matrices and the units of vectors. In Example 4.1.6e the matrix `Mb` is formed with numbers of rows and columns equal to the number of values (that is lines) of the texts `Tsa` and `Tsb`.

Example 4.1.6e

```
TEXT [NVALUES=6] Ta; VALUES=!T(a,b,c,d,e,f)
& [NVALUES=4] Tb; VALUES=!T(g,h,i,j)
& [NVALUES=3] Tsa; VALUES=!T(d,a,f)
& Tsb; VALUES=!T(i,h,j)
MATRIX [ROWS=Ta; COLUMNS=Tb] Ma; VALUES=!(1...24)
CALCULATE Mb = Ma$[Tsa; Tsb]
PRINT Ma,Mb; FIELDWIDTH=6; DECIMALS=0
```

```
          Ma
  Tb       g      h      i      j
  Ta
   a       1      2      3      4
   b       5      6      7      8
   c       9     10     11     12
   d      13     14     15     16
   e      17     18     19     20
   f      21     22     23     24

          Mb
           1      2      3

   1      15     14     16
   2       3      2      4
   3      23     22     24
```

You can put in a missing identifier (`*`) to mean the complete set of elements from the dimension concerned. Example 4.1.6f shows how to transfer the values from columns 1 and 2 of the matrix `Ma` into the variates `Vc1` and `Vc2` respectively. Using qualified identifiers for transferring rows and columns of matrices to and from variates is more straightforward than using the `EQUATE` directive (4.3). The missing identifier (`*`) in the first qualifier for `Ma` indicates that Genstat is to take all the rows.

Example 4.1.6f

```
   2  MATRIX [ROWS=5; COLUMNS=4] Ma; VALUES=!(1...20)
   3  VARIATE [NVALUES=5] Vc1,Vc2
   4  CALCULATE Vc1,Vc2 = Ma$[*; 1,2]
   5  PRINT Ma; FIELDWIDTH=6; DECIMALS=0

                Ma
                 1      2      3      4

          1      1      2      3      4
          2      5      6      7      8
          3      9     10     11     12
          4     13     14     15     16
          5     17     18     19     20

   6  & Vc1,Vc2; FIELDWIDTH=6; DECIMALS=0

   Vc1    Vc2
     1      2
     5      6
     9     10
    13     14
    17     18
```

Single values from a qualified variate are treated as scalars, but those from the various types of matrices have the same type as their parent. If you want these one-by-one matrices to be used as scalars, you can include an embedded assignment in the expression. For example, to multiply the variate Va by the value in row 2 and column 1 of the matrix Ma, you should put:

```
SCALAR Sca
CALCULATE Vb = Va * (Sca = Ma$[2;1])
```

If you tried to use the expression `Va*Ma$[2;1]`, you would get an error message, since Genstat would object to multiplying the variate `Va` by the one-by-one matrix `Ma$[2;1]`.

4.2 Functions for use in expressions

This section lists and describes the functions that can be used in expressions. The general form is illustrated by the statement:

```
    CALCULATE y = LOG10(x)
```

Here `LOG10` is the name of a function, and the identifier enclosed in brackets is its argument. Throughout this section we use lower case for identifiers that are arguments or results of functions, such as `x` and `y` above, to contrast with the upper case conventionally used in this Guide for function names, such as `LOG10`.

The argument of a function can be a list of identifiers, or even an expression. Some functions may need two arguments, in which case the arguments are separated by a semicolon (`;`). For example:

```
    CALCULATE w = SORT(x; y+z)
```

(For an explanation of `SORT`, see below.) Genstat checks that you have given the correct number of arguments. With some functions, you do not need to set the second and subsequent arguments; in that case, you should omit the semicolons that would follow the last argument that you do use.

The functions in Genstat are divided into classes as follows: general and mathematical functions (4.2.1), scalar functions (4.2.2), variate functions (4.2.3), matrix functions (4.2.4), table functions (4.2.5), dummy functions (4.2.6), character functions (4.2.7), elements of structures (4.2.8), statistical functions (4.2.9), data and time functions (4.2.10), tree functions (4.2.11), graphics functions (4.2.12) and image functions 4.2.13). They are described in alphabetical order within each section. At the beginning of each class we set out the valid types of argument for each function, and the type of the result. We give synonyms, and abbreviations for the function names where these have fewer than four letters: for example, the matrix function `INVERSE` has the two abbreviations `INV` and `I`. You can abbreviate any function to four characters (1.7.1): for example, `LOG10` could be written as `LOG1` - although this particular abbreviation might be a little misleading! If more characters are given, Genstat checks up to and including the 32nd (but few if any functions have names that long).

Some operations are provided by directives and procedures instead of by functions. These are described in later sections of this chapter.

4.2.1 General and mathematical functions

In this subsection, `x`, `y`, `a`, `b`, `l` or `u` represent identifiers, or lists of identifiers, of any structures containing numerical data: that is, scalars, variates, factors, tables, matrices, diagonal matrices or symmetric matrices; `s` represents a scalar, `f` a factor and `v` a variate. Where `x` and `y` occur together as arguments they must be of the same type. Apart from `NEWLEVELS`, which produces a variate from a factor, the result of any of these functions has the same type as that of the first argument.

`ABS(x)`	gives the absolute value of x: $|x|$.
`ACOS(x)` or `ARCCOS(x)`	gives the inverse cosine of x ($-1 \le x \le 1$), with the result in radians.
`ANGLE(y; x)`	gives the inverse tangent of y/x, result in radians in range $(-\pi,\pi]$.
`ASIN(x)` or `ARCSIN(x)`	gives the inverse sine of x ($-1 \le x \le 1$), result in radians.
`ATAN(x)` or `ARCTAN(x)`	gives the arctangent (inverse tangent) of x, result in radians.
`BETA(a; b; x)`	Beta function B(a,b) or, if x is set, regularized incomplete Beta function I(a,b,x).
`BI0(x)`	modified Bessel function of the first kind I_0(x).
`BI1(x)`	modified Bessel function of the first kind I_1(x).
`BIN(x;n)`	modified Bessel function of the first kind I_n(x; n).

`BJ0(x)`	Bessel function of the first kind J_0(`x`).
`BJ1(x)`	Bessel function of the first kind J_1(`x`).
`BJN(x;n)`	Bessel function of the first kind J^n(`x`; `n`).
`BK0(x)`	modified Bessel function of the second kind K_0(`x`).
`BK1(x)`	modified Bessel function of the second kind K_1(`x`).
`BKN(x;n)`	modified Bessel function of the second kind K_n(`x`; `n`).
`BOUND(x; l; u)`	sets values of `x` less than `l` to `l`, and values greater than `u` to `u`; missing values can be set in `l` or `u` to imply no boundary.
`BY0(x)`	Bessel function of the second kind Y_0(`x`).
`BY1(x)`	Bessel function of the second kind Y_1(`x`).
`BYN(x;n)`	Bessel function of the second kind Y_n(`x`; `n`).
`CEILING(x)`	ceiling of `x`: returns for each value x_j of `x` the least integer *i* such that $i \geq x_j$.
`CIRCULATE(x; s)`	treats `x` as a circular list and shifts its values round the list according to the value and sign of `s`. For example, if `x` contains 1,2,3,4,5, and `s` is −2, then the result is 3,4,5,1,2; if `s` were 2, the result would be 4,5,1,2,3. If you omit the second operand, `CIRCULATE` moves the values by one place to the right: that is, `s`=1.
`COS(x)`	gives the cosine of `x`, for `x` in radians.
`COSH(x)`	hyperbolic cosine of `x, for x` in radians.
`CUMULATE(x)` or `CUM(x)`	forms the cumulative sum of the values of `x`: for example, the result from `x` with values 1,5,4 is 1,6,10. If the operand is a scalar, the result is the value of the scalar.
`DEGREES(x)`	converts angles `x` from radians to degrees.
`DIFFERENCE(x; s)`	forms the differences between consecutive elements of `x`: that is, the *i*th element of the result is $x_i - x_{i-s}$. If you omit the second operand, first differences are formed (`s`=1). If $i-s<1$ or $i-s>n$, where *n* is the number of values of `x`, the *i*th element is set to missing.
`DIGAMMA(x)`	digamma function of `x`, Ψ(`x`).
`EXP(x)`	gives the exponential function of `x`: e^x.
`FACTORIAL(x)`	factorial of `x` (`x`!): the values in `x` must be non-negative, missing values are given for results that are too large to be stored.
`FLOOR(x)`	floor of `x`: returns for each value x_j of `x` the largest integer *i* such that $i \leq x_j$.
`FRACTION(x)`	fractional part of `x` i.e. `x-INTEGER(x)`.
`GAMMA(a; x)`	Gamma function, Γ(`a`) or, if `x` is set, lower incomplete Gamma function γ(`a`,`x`).
`INTEGER(x)` or `INT(x)`	gives the integer part of `x`: [`x`].
`LEVELS(f)`	forms a variate containing the levels of the factor `f`.
`LNFACTORIAL(x)`	log of `x`! for non-negative integer values `x`.
`LNGAMMA(x)`	log-Gamma function, $\log_e(\Gamma(x))$, for `x`>0.
`LOG(x)`	gives the natural logarithm of `x` (`x`>0).
`LOG10(x)`	gives the logarithm to base 10 of `x` (`x`>0).
`MODULO(x; y)`	form modulus of `x` to base `y`.
`MVINSERT(x; y)`	replaces values in `x` by missing value wherever the second identifier stores a non-zero value (representing the logical result *true*).
`MVREPLACE(x; y)`	replaces missing values in `x` with corresponding values from

	`y`. Elements with missing values in both `x` and `y` produce a warning message.
`NCOMBINATIONS(x; y)`	number of combinations of `y` objects taken from a set of size `x`.
`NEWLEVELS(f; x)`	forms a variate from the factor `f`; the variate `x` contains values to correspond to the levels, and should be of the same length as the number of levels of the factor. If the second argument `x` is omitted, the ordinals (1, 2...) are given. The result of this function is a variate of the same length as `f`. For an example see 4.1.2.
`NPERMUTATIONS(x; y)`	number of permutations of `y` objects taken from a set of size `x`.
`RADIANS(x)`	converts angles `x` from degrees to radians.
`RANK(x)`	ranks of the values in `x`.
`REVERSE(x)`	reverses the values of `x`: for example, the result from `x` with values 1,2,3 is 3,2,1.
`ROUND(x)`	rounds to nearest integer.
`SHIFT(x; s)`	shifts the values of `x` by `s` places (to the right or left according to the sign of `s`). This is not a circular shift, and so some positions lose values; these are replaced with missing values. That is, the *i*th element of the result is the value that was in element $i-s$ unless $i-s\le 0$.
`SIGN(x)`	sign of `x` (−1, 0 or 1 for `x`<0, `x`==0 or `x`>0 respectively).
`SIN(x)`	gives the sine of `x`, for `x` in radians.
`SINH(x)`	hyperbolic sine of `x`, for `x` in radians.
`SORT(x; y)`	sorts the elements of `x` into the order that would put the values of `y` into ascending order; the values of `y` are left unchanged. If the second argument is omitted, the values of `x` are sorted into ascending order. `x` can be the same structure as `y`. See below for an example.
`SQRT(x)`	gives the square root of `x` (`x`≥0).
`STANDARDIZE(x)`	standardizes the values of `x` to have mean zero and variance one.
`TAN(x)`	tangent of `x`, for `x` in radians.
`TANH(x)`	hyperbolic tangent of `x`, for `x` in radians.
`TRIGAMMA(x)`	trigamma function of `x`.

Example 4.2.1 illustrates the functions `DIFFERENCE`, `INTEGER`, `ROUND`, `MVREPLACE`, `SIN` and `SORT`. In the example of `SORT`, Genstat sorts the missing values in the variate `Vsa` to the beginning of the array; tied units like these are kept in their order of occurrence in the index vector (`Vsa`).

Example 4.2.1

```
  VARIATE [VALUES=-0.4,4.1,8.4,*,-1.6,5.7,-2.3] Va
  CALCULATE Vb = DIFFERENCE(Va; 2)
  PRINT Va,Vb; FIELDWIDTH=6; DECIMALS=1

    Va    Vb
  -0.4     *
   4.1     *
   8.4   8.8
     *     *
  -1.6 -10.0
   5.7     *
  -2.3  -0.7
```

```
   5  CALCULATE Iva = INTEGER(Va)
   6  & Rva = ROUND(Va)
   7  PRINT Va,Iva,Rva; FIELDWIDTH=6; DECIMALS=1

    Va    Iva   Rva
  -0.4   0.0   0.0
   4.1   4.0   4.0
   8.4   8.0   8.0
     *     *     *
  -1.6  -1.0  -2.0
   5.7   5.0   6.0
  -2.3  -2.0  -2.0

   8  VARIATE [VALUES=1,2,3,27.3,5,6,7] Vb
   9  CALCULATE Vc = MVREPLACE(Va; Vb)
  10  PRINT Vc; DECIMALS=2

          Vc
       -0.40
        4.10
        8.40
       27.30
       -1.60
        5.70
       -2.30

  11  CALCULATE Ve = SIN(Vc)
  12  PRINT Ve; FIELDWIDTH=8; DECIMALS=3

      Ve
  -0.389
  -0.818
   0.855
   0.827
  -1.000
  -0.551
  -0.746

  13  VARIATE [VALUES=3,1,*,*,1,4,7,4,*] Vsa
  14  & [VALUES=1...9] Vsb
  15  CALCULATE Vsc = SORT(Vsb; Vsa)
  16  PRINT Vsc; FIELDWIDTH=6; DECIMALS=0

   Vsc
     3
     4
     9
     2
     5
     1
     6
     8
     7
```

4.2.2 Scalar functions

The scalar functions generate a scalar result from other types of structure. Some of these functions calculate a summary value describing some aspect of the contents of the structure such as the maximum value, the median value, the mean, the variance or the area under a curve. Other functions allow you to copy attributes of the structure in the argument: for example, `NVALUES` gives the number of values. Finally, the `CONSTANTS` function, which has a single-valued text (or a string) as its argument, provides an easy and accurate way of specifying various scalar constants such as π and the value used by Genstat to represent missing values.

In this subsection, `x` again represents any numerical structure (scalar, variate, factor, rectangular matrix, symmetric matrix, diagonal matrix or table), `f` is a factor, and `m` is either a rectangular matrix, a symmetric matrix or a diagonal matrix; `y` is a structure of the same type as `x`. All the functions produce a scalar result from each structure in the argument list; all except

NMV and NOBSERVATIONS ignore missing values in the structure. Thus, the function MEAN is equivalent to SUM divided by NOBSERVATIONS, and the function NOBSERVATIONS is equivalent to NVALUES minus NMV.

Restrictions on a variate within a scalar function do not carry over to the expression outside.

AREA(y; x)	numerically integrates the curve running through the points specified by variates x and y using the trapezoidal method; x must be monotonically increasing or decreasing.
CONSTANTS(t) or C(t)	provides the value of various constants, according to the contents of the string in the single-valued text t: e (for a string of 'e'), π ('pi'), missing value ('*' or 'missingvalue'), the conversion factor by which to multiply radians to get degrees ('degrees'), the conversion factor by which to multiply degrees to get radians ('radians'), the number ε defined as the smallest number such that the calculation 1+ε is detectable on the computer as greater than one ('epsilon'), the number used to represent infinity e.g when defining axes for graphics ('infinity' or '+infinity'), and the number used represent minus infinity ('-infinity'). The string can be specified in either upper or lower case (or any mixture) and can be abbreviated just like the string settings of options such as PRINT.
CORRELATION(x; y)	if both x and y are specified, returns a scalar giving the correlation between the values of x and y; if y is omitted, CORRELATION is a matrix function which forms a matrix of correlations from a (symmetric) matrix of sums of squares and products (4.2.4).
COVARIANCE(x; y) or COV(x; y)	calculates the covariance between values of x and y.
GCONSTANTS(g)	provides type numbers of Genstat data structures. The string g can therefore be either 'scalar', 'factor', 'text', 'variate', 'matrix', 'diagonalmatrix', 'symmetricmatrix', 'table', 'asave', 'tsave', 'expression', 'formula', 'dummy', 'pointer', 'lrv', 'sspm', 'tsm', 'rsave', 'tree', or 'vsave'. It can be specified in either upper or lower case (or any mixture) and can be abbreviated just like the string settings of options such as PRINT.
KURTOSIS(x)	kurtosis of the non-missing values in x (centred around zero i.e. subtracting the expected value of 3 for a Normal distribution)
MAXIMUM(x) or MAX(x)	finds the maximum of the values of x.

MAXPOSITION(x)	finds the position of the first instance of the maximum value within x. For a variate this is the number of the unit containing the maximum. For a matrix the row of the maximum value can then be calculated as
	`row = INTEGER((MAXPOSITION(x)-1)/NROWS(x))+1`
	and the column as
	`col = MAXPOSITION(x) - NROWS(x) * (row-1)`
	For a symmetric matrix, the column is
	`col = INTEGER((SQRT(8*MAXPOSITION(x)+1)+1)/2)`
	and the row is

	`row = MAXPOSITION(x) - col*(col-1)/2`
`MEAN(x)`	gives the mean of the values of `x`.
`MEDIAN(x)` or `MED(x)`	finds the median of the values of `x`.
`MINIMUM(x)` or `MIN(x)`	finds the minimum of the values of `x`.

`MINPOSITION(x)`	finds the position of the first instance of the minimum value within `x`. For a variate this is the number of the unit containing the minimum. For a matrix the row of the minimum value can then be calculated as
	`row = INTEGER((MINPOSITION(x)-1)/NROWS(x))+1`
	and the column as
	`col = MINPOSITION(x) - NROWS(x)*(row-1)`
	For a symmetric matrix, the column is
	`col=INTEGER((SQRT(8*MINPOSITION(x)+1)+1)/2)`
	and the row is
	`row = MINPOSITION(x) - col*(col-1)/2`

`NLEVELS(f)`	gives the number of levels of factor `f`.
`NMV(x)`	counts the number of missing values in `x` (taking account of any restrictions applied by the `RESTRICT` directive: 4.4.1).
`NOBSERVATIONS(x)`	counts the number of observations (non-missing values) in `x` (taking account of any restrictions applied by the `RESTRICT` directive: 4.4.1).
`NVALUES(x)` or `NVRESTRICTED(x)`	gives the number of values of `x`, including missing values and taking account of any restrictions applied by the `RESTRICT` directive (4.4.1).
`NVUNRESTRICTED(x)`	number of values of `x` ignoring restrictions (i.e. gives the full length of `x`).
`PAREA(y; x)`	area of the polygon with vertices specified by `y` and `x`.
`RANGE(x)`	range of values in `x`, i.e. `MAX(x) - MIN(x)`.
`SD(x)`	standard deviation of the non-missing values in `x`.
`SEMEAN(x)`	standard error of the mean of non-missing values in `x`.
`SKEWNESS(x)`	skewness of the non-missing values in `x`.
`SUM(x)` or `TOTAL(x)`	gives the sum of the values in `x`.
`TYPE(x)`	gives the type number of the data structure `x`.
`VARIANCE(x)` or `VAR(x)`	gives the variance of the values in `x` (the divisor being the number of non-missing values in `x`, minus 1).

For example:

Example 4.2.2

```
VARIATE [VALUES=8,2,16,4,1,10,*,30] Va
" Med, Mn, Tot, Obs, and Nv are declared implicitly (as scalars). "
CALCULATE Med = MEDIAN(Va)
& Mn = MEAN(Va)
& Tot = SUM(Va)
& Obs = NOBSERVATIONS(Va)
& Nv  = NVALUES(Va)
PRINT Med,Mn,Tot,Obs,Nv; FIELDWIDTH=8; DECIMALS=2

     Med        Mn       Tot       Obs        Nv
    8.00     10.14     71.00      7.00      8.00

FACTOR [LEVELS=!(1,2,4,8)] Ff
CALCULATE Nl = NLEVELS(Ff)
PRINT Nl; FIELDWIDTH=6; DECIMALS=1
```

```
  N1
 4.0
```

The following functions are similar to the scalar functions, in that they produce summaries of the values in any numerical structure. However, they produce several summaries (in a variate) rather than a single value.

`PERCENTILES(x; p)`	percentiles (defined in variate `p`) of the values of `x`.
`QUANTILES(x; q)`	quantiles (defined in variate `q`) of the values of `x`.
`RMEANS(x;p;q)`	running means of `x` using a window around each unit that includes `p` preceding and `q` succeeding observations; `p` must be set, default for `q` is 0.
`RNOBSERVATIONS(x;p;q)`	number of observations contributing to computation of running mean or total involving `p` preceding and `q` succeeding observations about each unit of `x`; `p` must be set, default for `q` is 0.
`RTOTALS(x;p;q)`	running totals of `x` using a window around each unit that includes `p` preceding and `q` succeeding observations; `p` must be set, default for `q` is 0.
`RUNS(x)`	length of run of values up to each unit in `x`.

Other summaries, including standard errors of skewness and kurtosis, can be produced by the `DESCRIBE` procedure (2:2.1.1). Summaries of "circular" data such as wind directions can be produced by the `CDESCRIBE` procedure (2:2.1.2).

4.2.3 Variate functions

Variate functions produce summaries across a set of variates or a set of scalars. They each have a single argument, which is a pointer to the set of variates or scalars to be summarized. The variates in a set must all be of the same length. If any of them is restricted, that restriction is applied to all of them; if several are restricted, each restriction must be to the same set of units. For a set of variates the result of each function is a variate of the same length as the variates in the set, while for a set of scalars the result is a scalar. For example, if `p` points to the variates `X1`, `X2` and `X3`, each of length *n*, `VMEANS(p)` produces a variate of length *n*, whose *i*th unit contains the mean of the values in the unit *i* of `X1`, `X2` and `X3`.

All the functions except `VNMV` and `VNOBSERVATIONS` ignore missing values. Thus, the function `VMEANS` is equivalent to `VSUMS` divided by `VNOBSERVATIONS`, and the function `VNOBSERVATIONS` is equivalent to `VNVALUES` minus `VNMV`.

`VCORRELATION(p1; p2)`	gives the correlation, at every unit, between the values of the corresponding structures in pointers `p1` and `p2`.
`VCOVARIANCE(p1; p2)`	gives the covariance, at every unit, between the values of the corresponding structures in pointers `p1` and `p2`.
`VKURTOSIS(p)`	kurtosis of the non-missing values in each unit of the variates (or scalars) in `p` (centred around zero i.e. subtracting the expected value of 3 for a Normal distribution).
`VMAXIMA(p)`	finds the maximum of the values in each unit over the variates (or scalars) in pointer `p`.
`VMEANS(p)`	gives the mean of the non-missing values in each unit over the variates (or scalars) in pointer `p`.
`VMEDIANS(p)`	finds the median of the values in each unit of the variates (or scalars) in pointer `p`.
`VMINIMA(p)`	finds the minimum of the values in each unit of the variates (or scalars) in pointer `p`.

VNMV(p)	counts the number of missing values in each unit of the variates (or scalars) in pointer p.
VNOBSERVATIONS(p)	counts the number of observations (non-missing values) in each unit of the variates (or scalars) in pointer p.
VNVALUES(p)	gives the total number of values in each unit of the variates (or scalars) in pointer p: that is the number of variates (or scalars) in p.
VPERCENTILES(p;s)	calculates percentiles for the value supplied in scalar s, across the set of variates in pointer p.
VPOSITIONS(x; p)	Gives the suffix of the first vector in the pointer p containing the value in each unit of the variate or text x.
VQUANTILES(p;s)	calculates quantiles for the probability supplied in scalar s, across the set of variates in pointer p.
VRANGE(p)	range of values within the units of the variates in pointer p.
VSD(x)	standard deviation of the non-missing values in each unit of the variates (or scalars) in p.
VSEMEANS(x)	standard error of the mean of non-missing values in each unit of the variates (or scalars) in p.
VSKEWNESS(x)	skewness of the non-missing values in each unit of the variates (or scalars) in p.
VSUMS(p) or VTOTAL(p)	gives the sum of the non-missing values in each unit of the variates (or scalars) in pointer p.
VVARIANCES(p)	gives the variance of the non-missing values in each unit of the variates (or scalars) in pointer p.

Example 4.2.3

```
VARIATE [NVALUES=6] X,Y,Z; \
  VALUES=!(28,*,18,26,*,17),!(12,27,*,34,*,15),!(17,25,3(*),20)
& Min,Mean,Max,Obs,Nval,Tot
POINTER [VALUES=X,Y,Z] P
CALCULATE Min = VMINIMA(P)
& Mean = VMEANS(P)
& Max = VMAXIMA(P)
& Obs = VNOBSERVATIONS(P)
& Nval = VNVALUES(P)
& Tot = VTOTALS(P)
PRINT X,Y,Z,Min,Mean,Max,Obs,Nval,Tot; FIELDWIDTH=8; DECIMALS=1
```

```
       X         Y         Z       Min      Mean       Max       Obs      Nval       Tot
    28.0      12.0      17.0      12.0      19.0      28.0       3.0       3.0      57.0
       *      27.0      25.0      25.0      26.0      27.0       2.0       3.0      52.0
    18.0         *         *      18.0      18.0      18.0       1.0       3.0      18.0
    26.0      34.0         *      26.0      30.0      34.0       2.0       3.0      60.0
       *         *         *         *         *         *       0.0       3.0         *
    17.0      15.0      20.0      15.0      17.3      20.0       3.0       3.0      52.0
```

4.2.4 Matrix functions

These functions operate on the various types of matrix available in Genstat. The type of the resulting structure depends on the function concerned. For some of the functions you can specify a variate, which is treated as a rectangular matrix with one column. Any restriction on the variate is then ignored. (Remember that matrices cannot be restricted.) A *matrix* is a rectangular, symmetric or diagonal matrix structure; a *square matrix* is a rectangular matrix with the same number of rows as of columns.

BASE(i; n)	column matrix with n rows, value one in row i and zero elsewhere.

`COLBIND(x;y)`	joins matrices `x` and `y` side by side.
`COLCENTRE(x)`	centres the columns of matrix `x` by subtracting their means.
`COLMEANS(x)`	mean of the non-missing elements of each row of matrix `x`.
`COLNOBSERVATIONS(x)`	number of non-missing elements in each column of matrix `x`.
`COLSUMS(x)`	sum of the non-missing elements of each column of matrix `x`.
`COL1(n)`	column matrix of 1's with `n` rows.
`CORRELATION(x)` or `CORRMAT(x)`	forms a correlation matrix from a symmetric matrix `x` that contains sums of squares and products: the values of the resulting symmetric matrix `c` are formed by $c_{ij} = x_{ij} / \surd(x_{ii}x_{jj})$. Note, `CORRELATION` with two arguments, `x` and `y`, can also be used to produce the (scalar) correlation between the values in two structures (4.2.2).
`CHOLESKI(x)`	forms the Choleski decomposition of a symmetric matrix `x`; this produces a square matrix `L` such that `x` = `LL'` and such that upper off-diagonal elements are zero. The symmetric matrix `x` must be positive semi-definite.
`DETERMINANT(x)` or `DET(x)` or `D(x)`	forms the determinant of a symmetric matrix or a square matrix; the result is a scalar. Genstat uses the decomposition `x` = `LU`, and the determinant is defined to be $\Pi\{l_{ii}u_{ii}\}$.
`DIAGONAL(x; b)`	form a diagonal matrix from a variate `x`, or takes diagonal of a square, symmetric or diagonal matrix `x`; `b` may be set if `x` is a matrix, to request a banded diagonal matrix of order `b` (returned as a square matrix with the values off the bands set to zero).
`DPRODUCT(x; y)`	direct or Kronecker product of matrices `x` and `y`: `x` ⊗ `y`.
`DSUM(x; y)`	direct sum of matrices `x` and `y` (`x` ⊕ `y`); alternatively, if the second argument is omitted, `x` can be a pointer and the function then gives `x[1]` ⊕ `x[2]` ⊕ ... `x[n]`.
`EVALUES(x)`	eigenvalues of `x` (as a diagonal matrix).
`EVECTORS(x)`	eigenvectors of `x` (as a rectangular matrix).
`GINVERSE(x)`	Moore-Penrose generalized inverse of square, symmetric or diagonal matrix `x`.
`IDENTITY(n)`	identity matrix of order `n` (returned as a diagonal matrix).
`INVERSE(x)` or `INV(x)` or `I(x)`	forms the inverse of a non-singular square, symmetric or diagonal matrix; the result is a square, symmetric or diagonal matrix, according to the type of `x`. For a square matrix, Genstat uses Crout's method by forming the lower and upper triangular decomposition of the matrix, `x` = `LU`, and inverting `L` and `U` separately. Genstat uses the equivalent decomposition (Choleski) for symmetric matrices, which must be positive semi-definite.
`KRONECKER(x; y)`	synonym for `DPRODUCT`.
`LSVECTORS(x)`	matrix of vectors from the left-hand side of a singular-value decomposition of `x`.
`LTPRODUCT(x; y)`	forms the left transposed product of `x` and `y`: that is, the matrix product of the transpose of `x` with `y`, which can also be written `T(x)*+y`. The structures `x` and `y` can be matrices or variates. The number of rows of `x` must equal the number of rows of `y`. The result is a rectangular matrix with number

	of rows equal to the number of columns of `x` and number of columns equal to the number of columns of `y`, unless both `x` and `y` are diagonal matrices when the result is also a diagonal matrix.
`LTRIANGLE(m; d)`	returns the lower triangle of square matrix `m`, as a square matrix with the upper triangular set to zero; putting `d`=1 (default) indicates that the diagonal is to be included, while putting `d`=0 sets the diagonal to zero.
`MAT0(r; c)` or `MZERO(r; c)`	zero matrix of size `r` by `c`.
`MAT1(r; c)`	matrix of ones of size `r` by `c`.
`MBASE(r; c; i; j)`	matrix of size `r` by `c` which is zero, except for position(s) `i`,`j` which are set to one.
`MCENTRE(m)`	doubly centres matrix `m` so that its rows and columns have mean zero.
`MEXP(m)`	calculates the matrix exponential of `m`.
`MINSERT(x;m;i;j)`	inserts matrix `m` into matrix `x`, putting its top-left element into row `i` and column `j` of `x`; elements of `m` that are defined to lie outside `x` are ignored (so negative values of `i` and `j` are permitted).
`MPOWER(m; n)`	raises matrix `m` to the `n`'th power.
`MSQRT(m)`	calculates the matrix square root of `m`.
`NCOLUMNS(m)`	gives the number of columns of matrix `m`.
`NROWS(m)`	gives the number of rows of matrix `m`.
`PRODUCT(x; y)`	forms the matrix product of `x` and `y`; this can also be written `x*+y` using the operator *+. The structures `x` and `y` can be matrices or variates. The number of columns of `x` must equal the number of rows of `y`. The result is a rectangular matrix with number of rows equal to the number of rows of `x` and number of columns equal to the number of columns of `y`, unless both `x` and `y` are diagonal matrices when the result is also a diagonal matrix.
`QPRODUCT(x; y)`	forms the quadratic product of `x` and `y`; it can thus be written as `x*+y*+T(x)`, but the use of `QPRODUCT` is more efficient. `x` is a rectangular matrix or a variate, and `y` is a symmetric matrix or a diagonal matrix or a scalar. The number of columns of `x` must be the same as the number of rows of `y`. The result is a symmetric matrix with number of rows equal to the number of rows of `x`.
`QTPRODUCT(x; y)`	quadratic matrix product of `x'` and `y`: i.e. `QPRODUCT(TRANSPOSE(x);y)`.
`ROWBIND(x;y)`	joins matrices `x` and `y` vertically (i.e. stacks `y` below `x`).
`ROWCENTRE(x)`	centres the rows of matrix `x` by subtracting their means.
`ROWMEANS(x)`	mean of the non-missing elements of each row of matrix `x`.
`ROWNOBSERVATIONS(x)`	number of non-missing elements in each row of matrix `x`.
`ROWSUMS(x)`	sum of the non-missing elements of each row of matrix `x`.
`ROW1(n)`	row matrix of 1's with `n` columns.
`RSVECTORS(x)`	matrix of vectors from the right-hand side of a singular-value decomposition of `x`.
`RTPRODUCT(x; y)`	forms the right transposed product of `x` and `y`: that is, the matrix product of `x` with the transpose of `y`, which can also be written `x*+T(y)`. The structures `x` and `y` can be matrices

	or variates. The number of columns of x must equal the number of columns of y. The result is a rectangular matrix with number of rows equal to the number of rows of x and number of columns equal to the number of rows of y, unless both x and y are diagonal matrices when the result is also a diagonal matrix.
`SOLUTION(x; y)`	solves a set of simultaneous linear equations x*+b=y: $x_{11} b_1 + x_{12} b_2 + ... + x_{1n} b_n = y_1$ ... $x_{n1} b_1 + x_{n2} b_2 + ... + x_{nn} b_n = y_n$ The function thus finds b, as in the alternative expression `CALCULATE b = PRODUCT(INVERSE(x); y)` but the use of SOLUTION is more efficient and numerically stable than using PRODUCT and INVERSE: x is a square matrix and y is a rectangular matrix or a variate. The number of rows of x must be the same as the number of rows of y. The result is a rectangular matrix with numbers of rows and columns the same as y.
`SUBMAT(x)`	forms sub-triangles or sub-rectangles of a rectangular or symmetric matrix x, whose dimensions must be labelled by pointers. The structure to receive the values must have been declared already, as a rectangular or symmetric matrix according to the type of x, and have each of its dimensions also labelled by a pointer whose values are included in the pointer of the corresponding dimension of x. The correspondence between the values of the pointers that label the resulting matrix and those labelling x determines which rows and columns of x appear in the result. The same effect can be obtained by using the function ELEMENTS with a single list or expression for symmetric matrices, and with two lists for rectangular matrices. Just as with the ELEMENTS function, the resulting matrix can be made larger than x, by specifying repeated identifiers in its pointers.
`SVALUES(x)`	singular values of x (as a diagonal matrix).
`TRACE(x)`	forms the trace of matrix x: that is, the sum of its diagonal elements. x can be a square matrix, a diagonal matrix or a symmetric matrix. The result is a scalar.
`TRANSPOSE(x)` or `T(x)`	forms the transpose of x, where x is a rectangular matrix or a variate. The result is a rectangular matrix.
`UTRIANGLE(m; d)`	returns the upper triangle of square matrix m as a square matrix with the lower triangular set to zero; putting d=1 (default) indicates that the diagonal is to be included, while putting d=0 sets the diagonal to zero.
`VEC(x)`	stacks columns of a matrix x into a single variate (*VEC* operator).
`VECH(x)`	stacks columns of the lower triangle of a matrix x (*VECH* operator).

Example 4.2.4

```
  2  SYMMETRICMATRIX [ROWS=4] Sma; \
  3    VALUES=!(36,40,64,65,90,144,80,110,175,225)
  4  MATRIX [ROWS=4; COLUMNS=4] Chsma
  5  CALCULATE  Chsma = CHOLESKI(Sma)
  6  PRINT Chsma; FIELDWIDTH=8; DECIMALS=3

                Chsma
                    1       2       3       4

            1   6.000   0.000   0.000   0.000
            2   6.667   4.422   0.000   0.000
            3  10.833   4.020   3.237   0.000
            4  13.333   4.774   3.511   3.479

  7  MATRIX [ROWS=3; COLUMNS=3] Ma; VALUES=!(1,1,2,3,4,5,1,4,2)
  8  & Mainv
  9  CALCULATE Mainv = INVERSE(Ma)
 10  PRINT Ma; FIELDWIDTH=8; DECIMALS=3

                   Ma
                    1       2       3

            1   1.000   1.000   2.000
            2   3.000   4.000   5.000
            3   1.000   4.000   2.000

 11  & Mainv; FIELDWIDTH=8; DECIMALS=3

                Mainv
                    1       2       3

            1  -4.000   2.000  -1.000
            2  -0.333   0.000   0.333
            3   2.667  -1.000   0.333

 12  MATRIX [ROWS=3; COLUMNS=3] Mx; VALUES=!(1,1,2,3,4,5,1,4,2)
 13  & [ROWS=3; COLUMNS=1] My; VALUES=!(4,5,6)
 14  & Bxy
 15  CALCULATE Bxy = SOLUTION(Mx; My)
 16  PRINT Bxy; FIELDWIDTH=8; DECIMALS=3

                  Bxy
                    1

            1 -12.000
            2   0.667
            3   7.667

 17  VARIATE Va,Vb,Vc,Vd,Ve,Vf,Vg,Vh,Vi,Vj
 18  POINTER Pa,Pb,Pc,Pd; VALUES=!P(Va,Vb,Vc,Vd,Ve,Vf),!P(Vg,Vh,Vi,Vj), \
 19    !P(Vc,Va,Vf,Ve),!P(Vi,Vh,Vg)
 20  MATRIX [ROWS=Pa; COLUMNS=Pb] Ma ; VALUES=!(1...24)
 21  & [ROWS=Pc; COLUMNS=Pd] Mb
 22  CALCULATE Mb = SUBMAT(Ma)
 23  PRINT Ma; FIELDWIDTH=8; DECIMALS=1

                   Ma
           Pb      Vg      Vh      Vi      Vj
           Pa
           Va     1.0     2.0     3.0     4.0
           Vb     5.0     6.0     7.0     8.0
           Vc     9.0    10.0    11.0    12.0
           Vd    13.0    14.0    15.0    16.0
           Ve    17.0    18.0    19.0    20.0
           Vf    21.0    22.0    23.0    24.0

 24  & Mb; FIELDWIDTH=8; DECIMALS=1
```

```
           Mb
Pd         Vi        Vh        Vg
Pc
Vc       11.0      10.0       9.0
Va        3.0       2.0       1.0
Vf       23.0      22.0      21.0
Ve       19.0      18.0      17.0
```

There are several functions for forming matrices from tables.

`TCOLUMN(t)`	converts one-way table `t` into a matrix with a single column.
`TDIAGONAL(t)`	converts one-way table `t` into a diagonal matrix.
`TMATRIX(t; f1; f2)`	converts two-way table `t` into a matrix, with classifying factor `f1` corresponding to the rows, and classifying factor `f2` corresponding to the columns.
`TROW(t)`	converts one-way table `t` into a matrix with a single row.

Other matrix operations and decompositions are described in 4.10.

4.2.5 Table functions

The table functions operate on tables to produce new values for extended or summarized tables; for example,

```
    CALCULATE tr = TMEANS(ta)
```

takes means of certain of the cells in table `ta` and puts them in the table `tr`. If the resulting table, `tr` above, has already been declared, it must have the same status for margins as the corresponding table in the function (`ta` above). But if `tr` is left to be declared implicitly, it will be given margins whether or not they occur in `ta`. Summaries are produced over the levels of the factors that occur in `ta` but not in `tr`; the type of summary depends on which function is used. Then, if there are factors that occur in `tr` but not in `ta`, these are given duplicate values as described in 4.1.4. Finally, if tr has margins, these are filled in according to the function specified. For example, if `tr` is classified by factors `A` and `B` but `ta` is classified by `A`, `B` and `C`,

```
    CALCULATE tr = TMEANS(ta)
```

will put, in each cell of `tr`, means over the levels of factor `C`, as shown in Example 4.2.5.

`TKURTOSIS(x)`	forms margins containing the kurtosis of the values in table `t` (centred around zero i.e. subtracting the expected value of 3 for a Normal distribution).
`TMAXIMA(t)`	forms margins of maxima for table `t`.
`TMEDIANS(t)`	forms margins of medians for table `t`.
`TMEANS(t)`	forms margins of means for table `t`.
`TMINIMA(t)`	forms margins of minima for table `t`.
`TNOBSERVATIONS(t)`	forms margins counting the numbers of observations (non-missing values) in table `t`.
`TNMV(t)`	forms margins counting the numbers of missing values in table `t`.
`TNVALUES(t)`	forms margins counting the numbers of values, missing or non-missing, in table `t`.
`TSD(t)`	forms margins of standard deviations for table `t`.
`TSEMEANS(t)`	forms margins of standard errors for the (margins of) means of table `t`.
`TSKEWNESS(x)`	forms margins containing the skewness of the values in table `t`.

`TSUMS(t)` or `TTOTALS(t)`	forms margins of totals for table `t`.
`TVARIANCES(t)`	forms margins of between-cell variances for table `t`.

Example 4.2.5

```
FACTOR [LEVELS=2] A,B,C
TABLE [CLASSIFICATION=A,B,C] Ta; VALUES=!(1...8)
& [CLASSIFICATION=A,B] Tr
CALCULATE Tr = TMEANS(Ta)
PRINT Ta,Tr; FIELDWIDTH=6; DECIMALS=1

                          Ta
                  C        1      2
     A            B
     1            1      1.0    2.0
                  2      3.0    4.0
     2            1      5.0    6.0
                  2      7.0    8.0

            Tr
     B       1      2
     A
     1     1.5    3.5
     2     5.5    7.5
```

The functions for copying tables into matrices are described in 4.2.4. The functions `TPROJECT`, and `TVECTOR`, which copy tables into variates are described in 4.2.8.

4.2.6 Dummy functions

The function `SET` and its converse `UNSET` allow you to check whether a dummy is set; these are useful particularly in procedures (5.3) and `FOR` loops (5.2.1).

`SET(x)`	returns a scalar logical value containing the values 1 or 0 according to whether or not dummy `x` is set.
`UNSET(d)`	gives a scalar logical value (0 or 1) indicating whether or not the dummy `d` is set: that is, whether or not `d` points to another structure (i.e. the opposite of the function `SET`).

4.2.7 Character functions

This subsection describes the functions in Genstat that allow you to obtain information about text structures. As already mentioned, in 4.2.2, you can ascertain the number of lines in a text using the `NVALUES` function, the number of missing lines (null strings) by `NMV`, and the number of non-missing lines by `NOBSERVATIONS`. The functions described here produce variates from a text, giving details of the contents of each of its lines.

The `CHARACTER` function indicates the length of each line of the text, while `GETFIRST` and `GETLAST` find the position of the first or last non-space character in each line respectively.

`GETPOSITION` lets you find the position, in each line of the text in its first argument, of the corresponding line from the text in its second argument. This implies that the lines from the second text are shorter than or equal to the lines of the first text. In addition, there is an optional third argument (a logical), which allows you to specify whether or not comparisons of characters/letters are case sensitive. The default is false (that is, 0), which means that comparisons are case sensitive. If the third argument is set to true (a non-zero value), either as a scalar or in a variate with the same number of values as there are lines in the first argument, then lower and upper case letters are treated as the same; that is, comparisons are case insensitive.

`CHARACTERS(t;x)`	returns a variate giving the length of each line of text `t`: if `x` is omitted or set to 0 the length is the "raw" length (with no checking for any typesetting commands); if `x` = 1 it is the formatted length (taking account of typesetting commands, see 1.4.2 for their syntax); finally, if `x` = `-1` it is the number of storage units ("bytes") required to store the text (standard characters like letters and digits require only one, more complicated characters like Chinese or Thai characters may require as many as four).
`GETFIRST(t)`	gives a variate containing the position of the first non-space character in each string of text `t`.
`GETLAST(t)`	gives a variate containing the position of the last non-space character in each string of the text `t`.
`GETPOSITION(t1; t2; x)`	for each unit, if the string in `t2` occurs as a substring of the string in `t1`, this returns the position at which the substring starts; otherwise it returns the value zero. `t2` may contain a single string to be checked against every string of `t1`. `x` can be either a scalar or a variate, and supplies a logical value to indicate whether to ignore the case of any letters; if `x` is omitted the logical is assumed to be false (case not ignored).

4.2.8 Elements of structures

The `ELEMENTS` function has a similar role to qualified identifiers (4.1.6). Two functions, `EXPAND` and `RESTRICTION`, are available to derive sets of values from the results of a `RESTRICT` statement (4.4.1). `POSITION` allows you to determine the position at which the values of one vector occur within another. `NEXPAND` provides a quick way of replicating the values of a structure, `UNIQUE` forms the unique values from within a structure, and `WHERE` (synonym `WHICH`) locates those that are logically true i.e. non-zero. `TPROJECT` is a specialized function that "projects" the values from a table into a variate with length equal to the length of the classifying factors of the table. The value in each unit of the variate is the table value corresponding to the values of the classifying factors on that unit. So, for example, this provides a way of forming a variate of fitted values from a table of means. `TVECTOR` copies values from a table into a variate, allowing margins to be included or excluded, and a list of its classifying factors to be specified to define the order in which the values are taken.

`ELEMENTS(x; e1; e2)`	specifies a set of elements of `x`; `e1` and `e2` are expressions. As with qualified identifiers, you cannot specify elements of scalars or tables. You cannot use a text in any of the arguments of `ELEMENTS`. However the ability to specify expressions in the second and third arguments, instead of merely structures, is one way in which the use of `ELEMENTS` is more powerful that the use of qualified identifiers.
`EXPAND(x; s)`	forms a variate of zeroes and ones from the values of `x`, which Genstat takes to be a list of unit numbers; usually `x` will have been formed as the save structure from a `RESTRICT` statement. The second argument, `s`, is a scalar defining the length of the result; if `s` is omitted and `EXPAND` cannot determine the length of the result from its context within the expression, the resulting variate will take its length from the units structure (2.3.4).
`NEXPAND(n; v)`	expands structure `v` to repeat each value the number of times specified by the corresponding element of `n`.

`POSITION(x; y)`	finds the position, within the vector `y`, of each value of `x`.
`REPLACE(x;y;z)`	searches `x` for all occurrences of each value in `y`, and replaces them with the corresponding value from `z`.
`RESTRICTION(x)`	forms a variate with ones in the positions of the set of units to which `x` is currently restricted; the other units of the result are left unchanged (or left as missing values if no values have been set previously). If this variate is declared implicitly here, it will be restricted in the same way and have the same number of values as `x`. If you use the `RESTRICTION` function on its own in the `CONDITION` parameter of the `RESTRICT` directive (4.4.1), the restriction on `x` is passed to all the vectors listed with first parameter of `RESTRICT`.
`UNIQUE(x)`	the unique values in `x`.
`TPROJECT(t)`	converts table `t` into a variate, using the values of its classifying factors to determine which value of the table to put into each unit of the variate.
`TVECTOR(t; s; p)`	copies the values from table `t` into a variate. The scalar `s` is zero if the margins of the table are to be omitted, or a non-zero (and non-missing) value if they are included. The pointer `p` contains the classifying factors of the table, defining the order in which the values are to be copied; this can be omitted if `t` is a one-way table. If margins are not to be included from a one-way table, `s` can also be omitted.
`WHERE(x)` or `WHICH(x)`	produces a variate listing the units of `x` that are logically true, i.e. non-zero; if all the units are false, it produces a variate of length one, containing a missing value. If `x` is restricted or qualified (4.1.6), `WHERE` looks only in the defined subset of units for true values. However, the unit numbers in the result give their positions in the full vector `x`, not the subset.

The rules of dimensionality of the structures to which `ELEMENTS` is applied, and the specification of the expressions `e1` and `e2`, which identify the elements in each dimension, are similar to those for qualified identifiers (4.1.6). If `x` is a variate, a factor or a diagonal matrix, you should not specify the third argument `e2`; the type of the result is the same as that of `x`. You can also omit the third argument if `x` is a symmetric matrix, in which case the result is also a symmetric matrix; or you can specify both expressions, in which case the result is a rectangular matrix. For rectangular matrices, both `e1` and `e2` must be specified, and the result is a rectangular matrix. Genstat evaluates each expression and treats the result as a one-dimensional list of values. In line 5 of Example 4.2.8a, the values of the symmetric matrix `Smb` are taken from the rows and columns of the symmetric matrix `Sma` indicated by variate `Va`.

Example 4.2.8a

```
   2  SYMMETRICMATRIX [ROWS=5] Sma; VALUES=!(15...1)
   3  & [ROWS=3] Smb
   4  VARIATE Va; VALUES=!(5,4,2)
   5  CALCULATE Smb = ELEMENTS(Sma; Va)
   6  PRINT Sma,Smb; FIELDWIDTH=5; DECIMALS=0
```

```
     Sma

  1   15
  2   14   13
  3   12   11   10
  4    9    8    7    6
  5    5    4    3    2    1
       1    2    3    4    5

     Smb

  1    1
  2    2    6
  3    4    8   13
       1    2    3

  7  VARIATE Vb,Vc; VALUES=!(5,3,1),!(1,4,3)
  8  MATRIX [ROWS=3; COLUMNS=3; VALUES=1...9] Ma
  9  CALCULATE ELEMENTS(Sma; Vb; Vc) = Ma
 10  PRINT Sma; FIELDWIDTH=4; DECIMALS=0

    Sma

  1   7
  2  14  13
  3   9  11   6
  4   8   8   5   6
  5   1   4   3   2   1
      1   2   3   4   5
```

`ELEMENTS` is the only function that you are allowed to put on the left-hand side of an assignment. This is illustrated in line 9 of Example 4.2.8a, where the values of the matrix `Ma` are assigned to the elements of the symmetric matrix `Sma` indicated by the variates `Va` and `Vb`. Since `Sma` is symmetric, any values above the main diagonal indicated by `Va` and `Vb` are automatically transposed to their corresponding position below the diagonal.

Example 4.2.8b illustrates the use of the functions `EXPAND`, `NEXPAND`, `RESTRICTION`, `POSITION` and `UNIQUE`.

Example 4.2.8b

```
  2  VARIATE [VALUES=35,24,27,26,42,57] Age
  3  RESTRICT Age; CONDITION=Age>30; SAVESET=Va
  4  CALCULATE Vb = EXPAND(Va; 8)
  5  PRINT [ORIENTATION=across] Va,Vb; FIELDWIDTH=6; DECIMALS=0

          Va     1     5     6

          Vb     1     0     0     0     1     1     0     0

  6  VARIATE [VALUES=6(-1)] Rest
  7  CALCULATE Rest = RESTRICTION(Age)
  8  " Cancel the restriction on Age. "
  9  RESTRICT Age
 10  PRINT Age,Rest; FIELDWIDTH=6; DECIMALS=0

   Age  Rest
    35     1
    24     0
    27     0
    26     0
    42     1
    57     1

 11  VARIATE [VALUES=17,24,48,5] Vals
 12  & [VALUES=1,3,2,4] Reps
 13  CALCULATE Expanded = NEXPAND(Reps; Vals)
 14  & Unique = UNIQUE(Expanded)
 15  PRINT [ORIENTATION=across] Expanded; FIELD=3; DECIMALS=0
```

```
   Expanded 17 24 24 24 48 48  5  5  5  5

 16  PRINT Vals,Reps,Unique; DECIMALS=0

        Vals        Reps      Unique
          17           1           5
          24           3          17
          48           2          24
           5           4          48
```

Example 4.2.8c illustrates the functions `TPROJECT` and `TVECTOR`.

Example 4.2.8c

```
  2  FACTOR    [NVALUES=24; LEVELS=2] A
  3  FACTOR    [NVALUES=24; LEVELS=3] B
  4  FACTOR    [NVALUES=24; LEVELS=2] C
  5  GENERATE  2,A,B,C
  6  TABLE     [CLASSIFICATION=A,B,C; VALUES=101...112] T
  7  PRINT     T; DECIMALS=0

                                   T
                     C             1           2
          A          B
          1          1           101         102
                     2           103         104
                     3           105         106
          2          1           107         108
                     2           109         110
                     3           111         112


  8  CALCULATE Vp = TPROJECT(T)
  9  PRINT     A,B,C,Vp; DECIMALS=0

          A           B           C          Vp
          1           1           1         101
          1           1           2         102
          1           2           1         103
          1           2           2         104
          1           3           1         105
          1           3           2         106
          2           1           1         107
          2           1           2         108
          2           2           1         109
          2           2           2         110
          2           3           1         111
          2           3           2         112
          1           1           1         101
          1           1           2         102
          1           2           1         103
          1           2           2         104
          1           3           1         105
          1           3           2         106
          2           1           1         107
          2           1           2         108
          2           2           1         109
          2           2           2         110
          2           3           1         111
          2           3           2         112

 10  MARGIN    T; NEWTABLE=Tm; METHOD=totals
 11  PRINT     Tm
```

```
                               Tm
                     C          1           2      Margin
          A          B
          1          1      101.0       102.0       203.0
                     2      103.0       104.0       207.0
                     3      105.0       106.0       211.0
                Margin      309.0       312.0       621.0
          2          1      107.0       108.0       215.0
                     2      109.0       110.0       219.0
                     3      111.0       112.0       223.0
                Margin      327.0       330.0       657.0
     Margin          1      208.0       210.0       418.0
                     2      212.0       214.0       426.0
                     3      216.0       218.0       434.0
                Margin      636.0       642.0      1278.0

 12  CALCULATE Vt = TVECTOR(Tm; 0; !p(C,B,A))
 13  &         Vtm = TVECTOR(Tm; 1; !p(C,B,A))
 14  PRINT     Vt,Vtm; DECIMALS=0

          Vt         Vtm
         101         101
         107         107
         103         208
         109         103
         105         109
         111         212
         102         105
         108         111
         104         216
         110         309
         106         327
         112         636
                     102
                     108
                     210
                     104
                     110
                     214
                     106
                     112
                     218
                     312
                     330
                     642
                     203
                     215
                     418
                     207
                     219
                     426
                     211
                     223
                     434
                     621
                     657
                    1278
```

4.2.9 Statistical functions

The statistical functions cover various activities relevant to statistical analyses.

There are functions to transform percentage data: PROBIT, LOGIT, CLOGLOG (complementary log-log) and ANGULAR. The inverse transformations are also available: PROBIT, ILOGIT, ICLOGLOG and IANGULAR.

Cumulative lower and upper probabilities, and equivalent deviates are available for various probability distributions: Normal, F, chi-square, t, binomial, Poisson, hypergeometric, beta, gamma, lognormal, bivariate Normal and inverse Normal. In addition, point probabilities are provided for the discrete distributions (binomial, Poisson and hypergeometric). These functions all have a standard form: first a prefix (for example CL for cumulative lower probabilities) and then the name of the distribution. There are also various natural synonyms, such as NORMAL for CLNORMAL.

Log-likelihoods can be calculated for samples from either binomial, gamma, Normal or Poisson distributions using functions LLBINOMIAL, LLGAMMA, LLNORMAL and LLPOISSON respectively. (Note, these omit constant terms that depend on the data but not on the parameters.)

There are several functions, prefixed GR, for generating pseudo-random numbers or selecting random samples. The seed used for the generation is controlled by the SEED option of CALCULATE, and the underlying algorithm is a modified version of that presented by Wichman & Hill (1982). If the SEED option has its default value of zero on the first time that random numbers are used in a job, the seed is initialized automatically (using the current time on the system clock). A zero value subsequently in the job causes Genstat to use a default seed that continues the existing sequence of random numbers. To allow results to be reproduced, the current default seed can be saved by the GET directive (5.6.1), and reset by the SET directive (5.6.2). The same principles apply to URAND, which provides an alternative to GRUNIFORM for generating pseudo-random numbers from a uniform distribution on [0,1], but here the seed is set by the first argument of the function.

Unless otherwise stated in the descriptions below, the arguments of the functions can be any compatible numerical data structures. Any constraints on their possible values are given with each description. Except for the log-likelihood functions and the function URAND, the result is a structure of the same type, dimension and number of values as the structure in the first argument.

The log-likelihood functions produce a scalar result. Their first arguments must be variates. The second and third arguments can be scalars or variates; if they are variates, they must be of the same length as the variate in the first argument. The meaning of the second and third arguments is given with each description, as well as the form of the expression used to calculate the log-likelihood.

ANGULAR(%p) or ANG(%p)	provides the angular transformation: %p is a percentage with 0<%p<100. The function forms $x = (180/\pi) \times \text{arcsine}(\sqrt{(\%p/100)})$ and so the result x is in degrees 0<x<90.
CED	synonym of EDCHISQUARE.
CHISQ	synonym of CLCHISQUARE.
CLBETA(x; a; b)	cumulative lower probability for a beta distribution with parameters a and b.
CLBINOMIAL(x; n; p)	probability of x or fewer successes out of n binomial trials with probability of success p.
CLBVARIATENORMAL(x; y; r)	cumulative lower probability for a bivariate Normal distribution with means 0, variances 1 and correlation r.
CLCHISQUARE(x; df; c)	cumulative lower probability for a non-central chi-square distribution with noncentrality parameter c; if the third parameter c is omitted, it is assumed to be zero, giving the ordinary (central) chi-square distribution.
CLF(x; df1; df2; c)	cumulative lower probability for a non-central F distribution with degrees of freedom df1 and df2, and noncentrality parameter c; if the fourth parameter c is omitted, it is assumed to be zero, giving the ordinary (central) F distribution.
CLGAMMA(x; k; t)	cumulative lower probability for a gamma distribution with shape parameter k (kappa) and scale parameter t (theta).
CLHYPERGEOMETRIC(j; l; m; n)	probability of x or fewer positive samples out of a

	total sample of size `m` from a population of size `n` of which `l` are positive(hypergeometric distribution).
`CLINVNORMAL(x; m; v)`	cumulative lower probability for an inverse Normal (or inverse Gaussian) distribution with mean `m` and variance `v`.
`CLLOGNORMAL(x)`	cumulative lower probability for a lognormal distribution corresponding to a Normal distribution with mean 0 and variance 1.
`CLNORMAL(x; m; v)`	cumulative lower probability for a Normal distribution with mean `m` (default 0) and variance `v` (default 1).
`CLOGLOG(p)`	takes the complementary log-log transformation of the percentages `p` (0<`p`<100%).
`CLPOISSON(j; m)`	probability of value of `x` or less for a Poisson distribution with mean `m`.
`CLSMMODULUS(x; df; n)`	cumulative lower probability for a Studentized maximum modulus distribution with degrees of freedom `df` and number of means `n`.
`CLSRANGE(x; df; n)`	cumulative lower probability for a Studentized range distribution with degrees of freedom `df` and number of means `n`.
`CLT(x; df; c)`	cumulative lower probability for a non-central Student's t distribution with degrees of freedom `df` and noncentrality parameter `c`; if the third parameter `c` is omitted, it is assumed to be zero, giving the ordinary (central) t distribution.
`CLUNIFORM(x; a; b)`	cumulative lower probability for a uniform distribution on [`a`,`b`].
`CUBETA(x; a; b)`	cumulative upper probability for a beta distribution with parameters `a` and `b`.
`CUBINOMIAL(j; n; p)`	probability of more than `x` successes out of `n` binomial trials with probability of success `p`.
`CUBVARIATENORMAL(x; y; r)`	cumulative upper probability for a bivariate Normal distribution with means 0, variances 1 and correlation `r`.
`CUCHISQUARE(x; df; c)`	cumulative upper probability for a non-central chi-square distribution with noncentrality parameter `c`; if the third parameter `c` is omitted, it is assumed to be zero, giving the ordinary (central) chi-square distribution.
`CUF(x; df1; df2; c)`	cumulative upper probability for a non-central F distribution with degrees of freedom `df1` and `df2`, and noncentrality parameter `c`; if the fourth parameter `c` is omitted, it is assumed to be zero, giving the ordinary (central) F distribution.
`CUGAMMA(x; k; t)`	cumulative upper probability for a gamma distribution with shape parameter `k` (kappa) and scale parameter `t` (theta).
`CUHYPERGEOMETRIC(j; l; m; n)`	probability of more than `x` positive samples out of a total sample of size `m` from a population of size `n` of which `l` are positive(hypergeometric distribution).
`CUINVNORMAL(x; m; v)`	cumulative upper probability for an inverse Normal (or inverse Gaussian) distribution with mean `m` and variance `v`.
`CULOGNORMAL(x)`	cumulative upper probability for a lognormal distribution corresponding to a Normal distribution with mean 0 and variance 1.
`CUNORMAL(x; m; v)`	cumulative upper probability for a Normal distribution with

	mean m (default 0) and variance v (default 1).
CUPOISSON(j; m)	probability of a value greater than x for a Poisson distribution with mean m.
CUSMMODULUS(x; df; n)	cumulative upper probability for a Studentized maximum modulus distribution with degrees of freedom df and number of means n.
CUSRANGE(x; df; n)	cumulative upper probability for a Studentized range distribution with degrees of freedom df and number of means n.
CUT(x; df; c)	cumulative upper probability for a non-central Student's t distribution with degrees of freedom df and noncentrality parameter c; if the third parameter c is omitted, it is assumed to be zero, giving the ordinary (central) t distribution.
CUUNIFORM(x; a; b)	cumulative upper probability for a uniform distribution on [a,b].
EDBETA(p; a; b)	equivalent deviate corresponding to cumulative lower probability p for a beta distribution with parameters a and b.
EDBINOMIAL(p; n; bp)	equivalent deviate corresponding to cumulative lower probability p for a binomial distribution with n trials and probability of success bp (returns the smallest integer x such that the probability of up to x successes is greater than or equal to p).
EDCHISQUARE(p; df; c)	equivalent deviate corresponding to cumulative lower probability p for a non-central chi-square distribution with noncentrality parameter c; if the third parameter c is omitted, it is assumed to be zero, giving the ordinary (central) chi-square distribution.
EDF(p; df1; df2; c)	equivalent deviate corresponding to cumulative lower probability p for a non-central F distribution with degrees of freedom df1 and df2, and noncentrality parameter c; if the fourth parameter c is omitted, it is assumed to be zero, giving the ordinary (central) F distribution.
EDGAMMA(p; k; t)	equivalent deviate corresponding to cumulative lower probability p for a gamma distribution with shape parameter k (kappa) and scale parameter t (theta).
EDHYPERGEOMETRIC(p; l; m; n)	equivalent deviate corresponding to cumulative lower probability p for a hypergeometric distribution with samples of size m from a population of size n of which l are positive (returns the smallest integer x such that the probability of up to x successes is greater than or equal to p).
EDINVNORMAL(p; m; v)	equivalent deviate corresponding to cumulative lower probability p for an inverse Normal (or inverse Gaussian) distribution with mean m and variance v.
EDLOGNORMAL(p)	equivalent deviate corresponding to cumulative lower probability p for a lognormal distribution corresponding to a Normal distribution with mean 0 and variance 1.
EDNORMAL(p; m; v)	equivalent deviate corresponding to cumulative lower probability p for a Normal distribution with mean m (default 0) and variance v (default 1).
EDPOISSON(p; m)	equivalent deviate corresponding to cumulative lower probability p for a Poisson distribution with mean m (returns

	the smallest integer x such that the probability of up to x successes is greater than or equal to p).
EDSMMODULUS(p; df; n)	equivalent deviate corresponding to cumulative lower probability p for a Studentized maximum modulus distribution with degrees of freedom df and number of means n.
EDSRANGE(p; df; n)	equivalent deviate corresponding to cumulative lower probability p for a Studentized range distribution with degrees of freedom df and number of means n.
EDT(p; df; c)	equivalent deviate corresponding to cumulative lower probability p for a non-central Student's t distribution with degrees of freedom df and noncentrality parameter c; if the third parameter c is omitted, it is assumed to be zero, giving the ordinary (central) t distribution.
EDUNIFORM(p; a; b)	equivalent deviate corresponding to cumulative lower probability p for a uniform distribution on [a,b].
FED	synonym of EDF.
FPROBABILITY	synonym of CLF.
FRATIO	synonym of CLF.
GRBETA(n; a; b)	generates n pseudo-random numbers from a Beta distribution with parameters a and b.
GRBINOMIAL(n; t; p)	generates n pseudo-random numbers from a Binomial distribution with t trials and probability p.
GRCHISQUARE(n; df; c)	generates n pseudo-random numbers from a chi-square distribution with degrees of freedom df and non-centrality parameter c (default c=0).
GRF(n; df1; df2; c)	generates n pseudo-random numbers from an F distribution with df1 and df2 degrees of freedom, and non-centrality parameter c (by default c1=0).
GRGAMMA(n; k; t)	generates n pseudo-random numbers from a Gamma distribution with shape parameter k (kappa) and scale parameter t (theta).
GRHYPERGEOMETRIC(n; l; m; p)	generates n pseudo-random numbers from a Hypergeometric distribution representing the number of positive values or successes in samples of size m from a population of size p of which l are positive.
GRLOGNORMAL(n; m; v)	generates n pseudo-random numbers from a lognormal distribution such that log(x) has a Normal distribution with mean m and variance v.
GRNORMAL(n; m; v)	generates n pseudo-random numbers from a Normal distribution with mean m (default 0) and variance v (default 1).
GRPOISSON(n; m)	generates n pseudo-random numbers from a Poisson distribution with mean m.
GRSAMPLE(n; v; p)	forms a variate of size n by sampling with replacement from variate v with probabilities (or relative weights) p; if p is omitted, the probabilities are assumed to be equal; if v is omitted, sampling is from a variate containing the integers 1...n.
GRSELECT(n; v; r)	forms a variate of size n by sampling from a population; if r is omitted, the population contains just one of each element

	of the variate `v`; alternatively `r` can supply a variate defining the replication of the elements of `v` within the population (i.e. the population is then defined as `NEXPAND(r; v)`, see 4.2.8); if `v` is omitted, sampling is from a variate containing the integers 1...`n`.
`GRT(n; df; c)`	generates `n` pseudo-random numbers from a Student's t distribution with degrees of freedom `df` and non-centrality parameter `c` (default `c`=0).
`GRUNIFORM(n; a; b)`	generates `n` pseudo-random numbers from a uniform distribution on [`a`,`b`].
`IANGULAR(x)`	gives the inverse of the angular transformation (result in percentages).
`ICLOGLOG(x)`	gives the inverse of the complementary log-log transformation (result in percentages).
`ILOGIT(x)`	gives the inverse of the logit transformation (result in percentages).
`IPROBIT(x)`	gives the inverse of the probit transformation (result in percentages).
`LLBINOMIAL(x; n; p)` or `LLB(x; n; p)`	provides the log-likelihood function for the binomial distribution with sample size `n` and mean proportion `p` (`n` and `p` are scalars or variates): Σ { x Log(n p / x) + (n−x) Log(n (1−p) / (n−x)) }
`LLGAMMA(x; k; t)` or `LLG(x; k; t)`	provides the log-likelihood function for the gamma distribution with shape parameter `k` and scale parameter `t` (`k` and `t` are scalars or variates): Σ { k Log(x / t) − (x / t) − Log(Γ (k)) }
`LLNORMAL(x; m; v)` or `LLN(x; m; v)`	provides the log-likelihood function for the Normal distribution with mean `m` and variance `v` (`m` and `v` are scalars or variates): −½ Σ { Log(v) + (x−m)(x−m)/v }
`LLPOISSON(x; m)` or `LLP(x; m)`	provides the log-likelihood function for the Poisson distribution with sample size `m` (`m` is a scalar or a variate): Σ { x Log(m/x) + x − m }
`LOGIT(p)`	takes the logit transformation log(`p`/(100−`p`)) of the percentages `p` (0<`p`<100%).
`NED`	synonym of `EDNORMAL`.
`NORMAL`	synonym of `CLNORMAL`.
`PRBETA(x; a; b)`	probability density function for a beta distribution with parameters `a` and `b`.
`PRBINOMIAL(x; n; p)`	probability of `x` successes out of n binomial trials with probability of success `p`.
`PRCHISQUARE(x; df; c)`	probability density function for a non-central chi-square distribution with noncentrality parameter `c`; if the third parameter `c` is omitted, it is assumed to be zero, giving the ordinary (central) chi-square distribution.
`PRF(x; df1; df2; c)`	probability density function for a non-central F distribution with degrees of freedom `df1` and `df2`, and noncentrality parameter `c`; if the fourth parameter `c` is omitted, it is assumed to be zero, giving the ordinary (central) F distribution.
`PRGAMMA(x; k; t)`	probability density function for a gamma distribution with

	shape parameter `k` (kappa) and scale parameter `t` (theta).
`PRHYPERGEOMETRIC(j; l; m; n)`	probability of `x` successes out of a sample of `m` from a population of size `n` of which `l` are positive(hypergeometric distribution).
`PRINVNORMAL(x; m; v)`	probability density function for an inverse Normal (or inverse Gaussian) distribution with mean `m` and variance `v`.
`PRLOGNORMAL(x)`	probability density function for a lognormal distribution corresponding to a normal distribution with mean 0 and variance 1.
`PRNORMAL(x; m; v)`	probability density function for a Normal distribution with mean `m` (default 0) and variance `v` (default 1).
`PROBIT(p)`	takes the probit transformation of the percentages `p` (0<`p`<100%).
`PRPOISSON(j; m)`	probability of obtaining the value `x` for a Poisson distribution with mean `m`.
`PRSMMODULUS(x; df; n)`	probability density function for a Studentized maximum modulus distribution with degrees of freedom `df` and number of means `n`.
`PRSRANGE(x; df; n)`	probability density function for a Studentized range distribution with degrees of freedom `df` and number of means `n`.
`PRT(x; df; c)`	probability density function for a non-central Student's t distribution with degrees of freedom `df` and noncentrality parameter `c`; if the third parameter `c` is omitted, it is assumed to be zero, giving the ordinary (central) t distribution.
`PRUNIFORM(p; a; b)`	probability density function for a uniform distribution on [`a`,`b`].
`URAND(s1; s2)`	provides a uniform pseudo-random number generator, giving values in the range [0,1]. `s1` is a scalar which specifies the seed for the random numbers. `s2` is also a scalar; if you set this, the result is a variate of length equal to the value of the scalar. If you omit `s2`, the type of the result of `URAND` is determined from the context of the expression: that is from the type of the structure that is to receive the values that are generated; if the receiving structure has not been declared already, it will be declared implicitly as a variate with the length of the units structure (2.3.4)
`RQOBJECTIVE(y; d; p; t)`	returns the objective function from fitting a quantile linear regression with a response variate `y`, a design matrix `d`, a probability value specified by the scalar `p`, and using a tolerance defined by the scalar `t`. If the fourth argument is omitted, a default tolerance of 10^{-12} is used. For more details of quantile regression, see the `FRQUANTILES` directive. The objective is calculated as `SUM( r * (p - (r.LT.0)) )` where r is the variate of residuals from the fit.
`SSPLINE(y; x; df; p)`	fits a smoothing-spline of `y` on `x`, with `df` degrees of freedom or (if `df` is missing) smoothing parameter `p`.

Example 4.2.9 illustrates the functions `LLNORMAL`, `NED`, `PRPOISSON` and `CLPOISSON`.

Example 4.2.9

```
   2  Normal log-likelihood for X with mean 0.6 and variance 1.9 "
   3  VARIATE [VALUES=4.0,-3.5,-1.3,-2.8,1.9,2.5,0.3,-0.8,1.2,0.9] X
   4  CALCULATE Loglik = LLNORMAL(X; 0.6; 1.9)
   5  PRINT Loglik

      Loglik
      -16.72

   6  " Transform Pr to Normal equivalent deviates "
   7  VARIATE Pr; VALUES=!(0.1,0.45,*,0.2,0.83,-0.3,0.95)
   8  " There is an invalid value in unit 6; this is
  -9    given a missing value and a warning is printed. "
  10  CALCULATE Tran = NED(Pr)

**** G5W0001 **** Warning (Code CA 58). Statement 1 on Line 10
Command: CALCULATE Tran = NED(Pr)
Error in argument for distribution function.
Invalid P-value for ED function.

  11  PRINT Tran,Pr; FIELDWIDTH=8,10; DECIMALS=2,3

    Tran        Pr
   -1.28     0.100
   -0.13     0.450
       *         *
   -0.84     0.200
    0.95     0.830
       *    -0.300
    1.64     0.950

  12  " Calculate probabilities and cumulative probabilities
 -13    for a Poisson distribution with mean 2.5 "
  14  VARIATE [V=1...10] N
  15  CALCULATE Prob = PRPOISSON(N; 2.5)
  16  & Cumprob = CLPOISSON(N; 2.5)
  17  PRINT N,Prob,Cumprob; DECIMALS=0,3,3

           N        Prob     Cumprob
           1       0.205       0.287
           2       0.257       0.544
           3       0.214       0.758
           4       0.134       0.891
           5       0.067       0.958
           6       0.028       0.986
           7       0.010       0.996
           8       0.003       0.999
           9       0.001       1.000
          10       0.000       1.000
```

Some additional probability calculations are provided by procedures: `GRMULTINORMAL` generates multivariate Normal pseudo-random numbers, `GRMNOMIAL` generates multinomial pseudo-random numbers, and `EDDUNNETT` calculates equivalent deviates for Dunnett's simultaneous confidence interval around a control.

4.2.10 Date-time functions

These functions manipulate dates and times (stored in any numerical data structure).

`DAY(x)`	the day of month corresponding to date-time value `x`.
`MONTH(x)`	the month corresponding to date-time value `x`.
`YEAR(x)`	the year corresponding to date-time value `x`.
`WEEKDAY(x)`	the day of the week (where Monday is weekday 1) corresponding to data-time value `x`.

`TIME(h; m; s)`	constructs the time value (days and fractions of days) corresponding to `h` hours, `m` minutes and `s` seconds.
`HOURS(x)`	the number of hours during the day corresponding to `x` (i.e. the number of hours recorded on a 24 hour clock at date-time value `x`).
`MFRACTION(x;p;m)`	returns the period within a month that date-time value `x` belongs to; `p` is the length of the period (e.g. 5 for pentade, 10 for decade), and `m` is the starting month (default 1).
`MINUTES(x)`	the number of minutes during the hour corresponding to `x` (i.e. the number of minutes recorded on a clock at date-time value `x`).
`SECONDS(x)`	the number of seconds (including fraction of seconds) during the minute corresponding to date-time value `x`.
`NDAYINYEAR(x;m)`	the number of the day in year corresponding to date-time value `x`, and starting the year at the beginning of month `m` (default 1).
`NWEEKINYEAR(x;s)`	number of the week through the year for date-time value `x`. The default setting for `s` is `'iso'`; this uses the definition of ISO Standard IS-8601 (1988) in which any week (starting on Monday) that lies in more than one year is assigned a week number for the year in which most of its days occur. The alternative setting, `'simple'`, takes the first week of the year as the one containing 1st January.
`LEAPYEAR(x)`	returns 1 if the year corresponding to date-time value `x` is a leap year, 0 otherwise.
`DATE(d; m; y)`	constructs the date value corresponding to day `d`, month `m` and year `y`.
`CPUTIME(x)`	returns a scalar containing the currently used cpu time (argument `x` is ignored).
`NOW(x)`	returns a scalar containing the current date and time (argument `x` is ignored).

4.2.11 Tree functions

These functions allow you to navigate around a tree, or to obtain information about trees and their nodes. In the specifications below, `t` represents a tree, `x` represents any numerical structure, and `m` and `n` are scalars. Examples are given in Section 4.12.

`BBELOW(t; n; m)`	provides a variate containing numbers of all the nodes below node `n` of tree `t`; if `m`=1 this gives only the terminal nodes below `n`, otherwise it includes internal nodes as well.
`BBRANCHES(t; n)`	provides a variate containing the numbers of the branches taken on the path to node `n` in tree `t` (the result is of the same length as the results of the `BPATH` function, and includes missing value as the final element, corresponding to `n` itself).
`BDEPTH(t; x)`	calculates the depths of nodes `x` in tree `t`.
`BMAXNODE(t)`	provides the maximum node number in tree `t`.
`BNBRANCHES(t; x)`	provides the number of branches below nodes `x` in tree `t` (0 for any that are terminal nodes).
`BNEXT(t; x; y)`	finds the numbers of the nodes on branches `y` from nodes `x` in tree `t` (returning a missing value for any terminal node).
`BNNODES(t)`	provides the number of nodes in tree `t`.
`BPATH(t; n)`	provides a variate containing the numbers of the nodes on

	the branch to node `n` in tree `t` (includes `n` itself as the final element).
`BPREVIOUS(t; x)`	finds the numbers of the nodes immediately above nodes `x` in tree `t` (or a missing value if a node is the root of the tree).
`BSCAN(t; x)`	finds the numbers of the nodes immediately after nodes `x` in tree `t` in an standard branch-by-branch order that visits each node once (or a missing value for the node that is the last one in the tree).
`BTERMINAL(t; x)`	finds the next terminal nodes after nodes `x` in tree `t` (or a missing value after the last terminal node).

4.2.12 Graphics functions

These functions allow you to map between an RGB colour and its red, blue and green components; see 6.9.9 for more information. In the definitions, `x`, `y` and `z` represent any structure containing numerical data. The result is a structure of the same type as `x`.

`BLUE(x)`	calculates the blue components of the RGB colour values in `x`.
`GRAY(x)` or `GREY(x)`	calculates RGB colour values for the values on the gray (grey) scale in `x`.
`GREEN(x)`	calculates the green components of the RGB colour values in `x`.
`RED(x)`	calculates the red components of the RGB colour values in `x`.
`RGB(x)`	calculates RGB colour values from the red, green and blue components in `x`, `y` and `z`, respectively; these components must all be between 0 and 255.
`RGB(t)`	provides the RGB colour values of the standard Genstat colours in text `t`. The text can contain the string `'match'` in its second and subsequent units, to repeat the colour in the previous unit. It can also contain strings made up of three pairs of hexadecimal digits (00-FF) prefixed by `#`, `0x` or `0X`: i.e. `'#rgb'`, `'0xrgb'` or `'0Xrgb'` where `rgb` are pairs of hexadecimal digits 00-FF that define the red, green and blue intensities of the colour respectively.

4.2.13 Image functions

These functions operate on images, represented as matrices of `RGB` values. See `PEN` (6.9.7) and `DBITMAP` (6.5.1) for more details. They are based on algorithms in the ImgSource library supplied by Smaller Animals Software, Inc. For more information, see http://www.smalleranimals.com/.

`IMBRIGHTNESS(r;l;h;m)`	modifies the brightness of the RGB image in matrix `r`, setting pixels in each channel with brightness less than `l` (default 0) to 0 and those brighter than `h` (default 255) to 255; `m` defines the mode of adjustment (default 0 stretches brightness and 1 distributes brightness evenly across the range).
`IMCONTRAST(r;c;b)`	modifies contrast and brightness of the RGB image in matrix `r`; `c` ($-1 \leq c \leq 1$; default 0 i.e. no adjustment) defines the adjustment to the contrast, and `b` ($-1 \leq b \leq 1$; default 0 i.e. no adjustment) defines the adjustment to the brightmess.

IMGAMMA(r;g)	applies gamma correction g (g≥0; default 1.5) to the brightness of the RGB image in matrix r; g<1 decreases brightness, and g>1 increases brightness.
IMGRAYSCALE(r) or IMGREYSCALE(r)	convert the RGB image in matrix r to grey scale.
IMCREPLACE(r;c;d;t)	replaces colour c in the RGB image in matrix r with colour d, using tolerance t (default 0).
IMSIZE(r;w;h;m)	changes the size of the RGB image in matrix r to have width w and height h; m selects the algorithm to use to assign colours in the new image: 0 = box filter, 1 = triangle filter, 2 = Hamming filter, 3 = Gaussian filter, 4 = bell filter, 5 = B-spline filter, 6 = cubic 1 filter, 7 = cubic 2 filter, 8 = Lanczos3 filter, 9 = Mitchell filter, 10 = sinc filter, 11 = Hermite filter, 12 = Hanning filter, 13 = Catrom filter, 14 = fast area-average, 15 = area-average, 16 = bi-linear interpolation, 17 (default) = bi-cubic interpolation, 18 = nearest neighbour.
IMROTATE(r;a;b)	rotates the RGB image in matrix r; a is the angle in radians (default $\pi/2$); b is the background colour to put into the (blank) corners.
IMHFLIP(r)	performs a horizontal flip on the RGB image in matrix r.
IMVFLIP(r)	performs a vertical flip on the RGB image in a matrix r.
IMPUSH(r;x1;y1;x2;y2)	applies a point-to-point warp on the RGB image in matrix r, "pushing" point (x1, y1) to (x2, y2).
IMXSHEAR(r;x;b)	shears the RGB image in matrix r by moving the top of the image |x| pixels to the right (x>0) or left (x<0); the blank parts of the new image are given (background) colour b.
IMYSHEAR(r;y;b)	shears the RGB image in matrix r by moving the right-hand side of the image |y| pixels up or down; the blank parts of the new image are given (background) colour b.
IMMEDIANFILTER(r)	performs a median filter on the RGB image in a matrix r.
IMSATURATE(r;s)	adjusts the saturation level of the RGB image in matrix r according to the value of scalar s (default 1.1): when s>1 the saturation is increased, when 0<s<1 saturation is decreased, and when s<0 photo-negative is generated.
IMSHARPEN(r;s)	sharpens the RGB image in matrix r by the amount specified in scalar s (0<s<100; default 2).
IMUNSHARPEN(r;t;a;s)	applies an unsharp mask to the RGB image in matrix r: this first applies a Gaussian blur with standard deviation s; it then finds the difference between pixels in the blurred image and in the original and, if this is greater than t in each channel, it adds the amount specified by scalar a multiplied by the difference from the original value.
IMBLUR(r;b)	blurs the RGB image in matrix r by the amount specified in scalar b (0<b<100; default 2).
IMDESPECKLE(r)	despeckles the RGB image in matrix r.
IMGBLUR(r;s)	applies a Gaussian blur with standard deviation s to the RGB image in matrix r.
IMCEQUALIZE(r;l;u)	performs an independent histogram equalization of the colours in the RGB image in matrix r; scalar l specifies the lower threshold and scalar h specifies the upper threshold.

`IMBEQUALIZE(r;l;u)`	performs a histogram equalization of the brightness of the RGB image in matrix `r`; scalar `l` specifies the lower threshold and scalar `h` specifies the upper threshold.
`IMBSTRETCH(r;l;u;m)`	performs a histogram stretch of the brightness in the RGB image in matrix `r`; scalar `l` (default 0) specifies the percentage of pixels to set to 0 (i.e. black), scalar `h` (default 0) specifies the percentage of pixels to set to white, and scalar `m` (0≤`m`≤255; default 128) specifies the colour value in each channel to be set to the middle intensity.
`IMSSTRETCH(r;l;h;m)`	performs a histogram stretch of the saturation in the RGB image in matrix `r`; scalar `l` (default 0) specifies the percentage of pixels to set to 0 (i.e. black), scalar `h` (default 0) specifies the percentage of pixels to set to white, and scalar `m` (0≤`m`≤255; default 128) specifies the colour value in each channel to be set to the middle intensity.
`IMCSTRETCH(r;l;h;m)`	performs a histogram stretch of the individual colours in the RGB image in matrix `r`; scalar `l` (default 0) specifies the percentage of pixels to set to 0 (i.e. black), scalar `h` (default 0) specifies the percentage of pixels to set to white, and scalar `m` (0≤`m`≤255; default 128) specifies the colour value in each channel to be set to the middle intensity.
`IMLINE(r;x1;y1;x2;y2;c)`	draws a line from point (`x1`, `y1`) to (`x2`, `y2`) in colour `c` on the RGB image in matrix `r`.
`IMELLIPSE(r;cx;cy;hr;vr;c;cf;p)`	draws an ellipse with centre (`cx`, `cy`), horizontal radius `hx` (default 40), vertical radius `vr` (default 40), colour `cl`, fill colour `cf` (default 0) and opacity `p` (0≤`p`≤1, where 0 is transparent and the default of 1 is solid) on the RGB image in matrix `r`.
`IMTEXT(r;st;c;fh;y1;x1;y2;x2;ft)`	draws the text in string `st` with height `fh`, font `ft` and colour `c` within the bounding rectangle with top left corner at (`x1`, `y1`) and bottom right corner at (`x2`, `y2`) on the RGB image in matrix `r`.
`IMSTEXT(r;st;c;fh;y1;x1;y2;x2;ft;tr;sm)`	draws the text in string `st` with height `fh`, font `ft`, colour `c`, transparency `tr` and smoothness `sm` (`sm`=1 for none, or 2 or 4) within the bounding rectangle with top left corner at (`x1`, `y1`) and bottom right corner at (`x2`, `y2`) on the RGB image in matrix `r`.
`IMRECTANGLE(r;x1;y1;x2;y2;c)`	colours the rectangle with bottom left corner (`x1`, `y1`) and top right corner (`x2`, `y2`) in the RGB image in matrix `r` to be colour `c`.
`IMEMBOSS(r;b;t;a;e;d)`	embosses the RGB image in matrix `r`; matrix `b` specifies a "bump map" defining the peaks and valleys in the output image (typically this is a grey scale version of `r`); matrix `t` defines the texture to apply to the input matrix; scalar `a` gives the angle of the light source in radians; scalar `e` is the elevation of the light source in radians; scalar `d` defines the depth of the effect.
`IMOVERLAY(rt;rb;m;mp;p;x;y)`	overlays the RGB image in matrix `rt` over the RGB image in matrix `rb`; `m` controls how images are blended (0 = fast blend, 1 = slower, more accurate blend, 2 = pixels

combined with logical `AND`, 3 = pixels combined with logical `OR`, 4 = pixels combined with logical `XOR`, 5 = output pixel is maximum of top and bottom as in Photoshop "Lighten", 6 = output pixel is minimum of top and bottom as in Photoshop "Darken", 7 = output pixel is sum of top and bottom, 8 = output pixel is difference of top and bottom, 9 = if top > `mp`, output top, 10 = if top < `mp`, output top, 11 = absolute value of the difference of top and bottom, 12 = take top × bottom / maximum component, 13 = take top × bottom × ModeParameter / maxComponent, 14 = screen, 15 = define bottom to be bottom + top − mp, 16 = define bottom to bottom − top − mp, 17 = pixels combined with logical NAND, 18 = pixels combined with logical NOR, 19 = pixels combined with logical `NXOR/XNOR`, 20 = color dodge, 21 = color burn, 22 = soft dodge, 23 = soft burn, 24 = Photoshop "overlay", 25 = soft light, 26 = hard light, 27 = XFader reflect, 28 = XFader glow, 29 = XFader freeze, 30 = XFader heat); `p` defines the opacity of the blended image; and (`x`, `y`) specifies the position of bottom left-hand corner of the top image on the bottom image.

`IM3CONVOLUTION(r;f;i;cr;cg;cb;d)` applies the convolution filter in the 3×3 matrix `f` to the RGB image in matrix `r`; scalar `i` (default 1) defines the intensity parameter; scalars `cr`, `cg` and `cb` contain 0 or 1 (default) according to whether the red, green and blue channels, respectively, are to be modified. If the "feedback" defined by scalar `d` is 0 (default), the new value at each point is `i` multiplied by the sum of the values at the point and nearby points multiplied by the convolution matrix. Alternatively, if `d`=i (default), the new value at each point is calculated by taking (1-`i`) multiplied by the current value at the point, and then subtracting `i` multiplied by the sum of the values at the point and nearby points multiplied by the values in the convolution matrix.

`IMMCONVOLUTION(r;f;i;cr;cg;cb;m)` applies the convolution filter in matrix `f` to the RGB image in matrix `r` ; scalar `i` (default 1) defines the intensity parameter; scalars `cr`, `cg` and `cb` contain 0 or 1 (default) according to whether the red, green and blue channels, respectively, are to be modified. If the mode defined by scalar `m` is 0 (default), the new value at each point is `i` multiplied by the sum of the values at the point and nearby points multiplied by the convolution matrix. Alternatively, if `m`=1 (default), the new value at each point is the current value at the point minus i multiplied by the sum of the values at the point and nearby points multiplied by the values in the convolution matrix.

4.3 Operations on sets of values: copying, comparison and Boolean calculations

The EQUATE directive (4.3.1) provides general facilities for copying values from one set of data structures to another. For example, you may wish to copy the values from a one-way table into a variate, or from a matrix into a set of variates (one variate for each row, or for each column), or the other way round, from variates into a matrix. EQUATE can be used to append values from several data structures into a single one (see Example 4.3.1a); the only constraint is that the structures in the respective sets must all contain the same kind of values. Note, however, that the APPEND and STACK procedures, described in Sections 4.4.4 and 4.4.5, provide more convenient methods of appending values of vectors. They use EQUATE internally but, for example, automatically define the output structures to be the correct length and also allow you to form a group factor to indicate where each value came from. In Genstat *for Windows*, you can also append vectors using the spreadsheet facilities. There are also specialized functions for copying tables into matrices, described in Section 4.2.8.

The SETRELATE directive checks to see whether the distinct set of values in two structures are identical, or whether one is a subset of the other (4.3.2). The structures must again have values of the same kind (numbers, strings or identifiers), but need not be of the same type. Boolean set calculations can be performed on the contents of vectors or pointers using the SETCALCULATE directive (4.3.3), which is used by the menu for Calculations on Sets in Genstat *for Windows*. You can construct all the ways in which a set of objects can be partitioned into subsets of a specified size using the SETALLOCATIONS directive (4.3.4), and you can find where particular values occur in a data structure using the GETLOCATIONS directive (4.3.5).

4.3.1 Copying between sets of structures: the EQUATE directive

EQUATE directive

Transfers data between structures of different sizes or types (but the same modes i.e. numerical or text) or where transfer is not from single structure to single structure.

Options

OLDFORMAT = *variate*	Format for values of OLDSTRUCTURES; within the variate, a positive value *n* means take *n* values, -*n* means skip *n* values and a missing value means skip to the next structure; default * i.e. take all the values in turn
NEWFORMAT = *variate*	Format for values of NEWSTRUCTURES; within the variate, a positive value *n* means fill the next *n* positions, -*n* means skip *n* positions and a missing value means skip to the next structure; default * i.e. fill all the positions in turn
FREPRESENTATION = *string token*	How to interpret factor values (labels, levels, ordinals); default leve

Parameters

OLDSTRUCTURES = *identifiers*	Structures whose values are to be transferred; if values of several structures are to be transferred to one item in the NEWSTRUCTURES list, they must be placed in a pointer
NEWSTRUCTURES = *identifiers*	Structures to take each set of transferred values; if several structures are to receive values from one item in

the OLDSTRUCTURES list, they must be placed in a pointer

The general idea with EQUATE is that the values in the structures in the OLDSTRUCTURES list are copied into the structures in the NEWSTRUCTURES list. Each item in OLDSTRUCTURES list specifies a single data structure, or a single set of data structures, containing the values to be transferred. A single structure can be a factor, or a text, or any one of the structures that contain numbers (scalar, variate, rectangular matrix, diagonal matrix, symmetric matrix or table). If you want to give a set of structures you must put them into a pointer. As already mentioned, all the structures in the set must contain the same kind of values: that is, they must all be texts, or all factors, or must all contain numbers (but they need not all be the same kinds of numerical structure – they could, for example, be a mixture of variates and matrices).

The corresponding entry in the NEWSTRUCTURE list indicates where the transferred data are to be placed. It is either a single structure or a pointer to a set of structures; the structures must be of a type suitable to store the values to be transferred.

In Example 4.3.1a, information about the employees of a firm has been typed in series in two separate sections, and the statement in lines 20 and 21 copies them into one; for each employee there are three pieces of information – name, grade and hours.

Example 4.3.1a

```
   2  OPEN 'Employee.dat'; CHANNEL=2
   3  " Read values for the first 6 employees,
  -4    in series, into Name1, Grade1 and Hours1."
   5  TEXT [NVALUES=6] Name1
   6  FACTOR [NVALUES=6; LEVELS=3] Grade1
   7  VARIATE [NVALUES=6] Hours1
   8  READ [PRINT=data,errors; CHANNEL=2; SERIAL=yes] Name1,Grade1,Hours1

     1  Clarke Innes Adams Jones Day Grey :
     2  2 1 2 1 1 3 :
     3  45 51 40 46 44 40 :
   9  " Read values for the final 4 employees,
 -10    in series, into Name2, Grade2 and Hours2."
  11  TEXT [NVALUES=4] Name2
  12  FACTOR [NVALUES=4; LEVELS=3] Grade2
  13  VARIATE [NVALUES=4] Hours2
  14  READ [PRINT=data,errors; CHANNEL=2; SERIAL=yes] Name2,Grade2,Hours2

     4  Edwards Baker Hill Foster :
     5  2 2 3 1 :
     6  47 42 40 41 :
  15  " Use EQUATE to put information about all the employees
 -16    into single vectors Name, Grade and Hours."
  17  TEXT [NVALUES=10] Name
  18  FACTOR [NVALUES=10; LEVELS=3] Grade
  19  VARIATE [NVALUES=10] Hours
  20  EQUATE !P(Name1,Name2),!P(Grade1,Grade2),!P(Hours1,Hours2); \
  21    NEWSTRUCTURES=Name,Grade,Hours
  22  PRINT Name,Grade,Hours

    Name       Grade       Hours
  Clarke           2       45.00
   Innes           1       51.00
   Adams           2       40.00
   Jones           1       46.00
     Day           1       44.00
    Grey           3       40.00
 Edwards           2       47.00
   Baker           2       42.00
    Hill           3       40.00
  Foster           1       41.00
```

Except with a format (see below) Genstat ignores where each structure within a set from the OLDSTRUCTURES list ends and another one begins: that is, it treats the set as being a concatenated list of values. Similarly, it treats the structures in each NEWSTRUCTURES set as an unstructured list of positions that are to receive values. The old values are repeated as often as is necessary to traverse all the new positions. Example 4.3.1b forms a matrix M with repeated and alternating rows taken from variates R1 and R2.

Example 4.3.1b

```
  2  VARIATE [VALUES=1...6] R1
  3    & [VALUES=101...106] R2
  4  " Form a matrix M whose rows are R1, R2, R1 and R2."
  5  MATRIX [ROWS=4; COLUMNS=6] M
  6  EQUATE !P(R1,R2); NEWSTRUCTURES=M
  7  PRINT M; FIELDWIDTH=6; DECIMALS=0

                M
                1     2     3     4     5     6

          1     1     2     3     4     5     6
          2   101   102   103   104   105   106
          3     1     2     3     4     5     6
          4   101   102   103   104   105   106
```

You can use the OLDFORMAT and NEWFORMAT options to control how the old values and new positions are traversed. The setting for each of these is a variate whose values are interpreted as follows:

(a) a positive integer *n* means take the next *n* values (OLDFORMAT) or fill the next *n* positions (NEWFORMAT);

(b) a negative integer -*n* means skip the next *n* values or positions;

(c) a missing value * means skip to the end of the structure.

As usual, Genstat recycles when it runs out of values. That is, if the contents of one of the variates is exhausted before all the NEWSTRUCTURES positions have either been filled or skipped, then that variate is repeated.

For example:

Example 4.3.1c

```
  8  "Form variates C[1...6] containing the values in the columns of M."
  9  VARIATE [NVALUES=4] C[1...6]
 10  EQUATE [OLDFORMAT=!((1,-5)4,-1)] M; NEWSTRUCTURES=C
 11  PRINT C[1...6]; FIELDWIDTH=6; DECIMALS=0

 C[1]  C[2]  C[3]  C[4]  C[5]  C[6]
    1     2     3     4     5     6
  101   102   103   104   105   106
    1     2     3     4     5     6
  101   102   103   104   105   106
```

This gives the variates C[1...6] the values in the columns of M. It does it by taking one column at a time from M, skipping the values in the other columns. (Remember that the values of M are held row-by-row.) In detail, what happens is this. For C[1], the format !((1,-5)4,-1) in line 10 takes the value in row 1 column 1, then skips the elements in the remaining five columns of row 1 before taking the value from column 1 of row 2. For C[1] this continues for each row of M, until the final element of the format, -1, skips column 1 of row 1, so that C[2] is given the values in column 2, and so on.

Notice that, as pointer C is automatically available to refer to C[1...6] (see 2.6), there is no need to put, for example, `!P(C[1...6])`.

The final part of the example shows how to form a matrix from a set of variates that contain the values for the columns.

Example 4.3.1d

```
  12  "Reform values of M so that its columns are C[1...6] in reverse order."
  13  EQUATE [OLDFORMAT=!((1,-3)6,-1)] !P(C[6...1]); NEWSTRUCTURES=M
  14  PRINT M; FIELDWIDTH=6; DECIMALS=0

                M
                1     2     3     4     5     6

          1     6     5     4     3     2     1
          2   106   105   104   103   102   101
          3     6     5     4     3     2     1
          4   106   105   104   103   102   101
```

If you are transferring values between factors, Genstat will check that each value to be transferred is valid for the factor in the `NEWSTRUCTURES` list. By default, Genstat will try to match the values using the levels of the factors, but you can set option `FREPRESENTATION` to `labels` to match by their labels, or to `ordinals` to match them merely according to the ordinal position in the levels vector of each factor. Example 4.3.1e illustrates the various possibilities.

Example 4.3.1e

```
  2  FACTOR [LEVELS=!(2,4); LABELS=!T(standard,double); VALUES=2,4,4,2]\
  3    Dose1
  4  FACTOR [NVALUES=8; LEVELS=3; LABELS=!T(none,standard,double)] Dose2
  5  " Form Dose2 from Dose1, repeated twice, matching by labels."
  6  EQUATE [FREPRESENTATION=labels] Dose1; NEW=Dose2
  7  PRINT [SERIAL=yes; ORIENTATION=across; RLWIDTH=6] \
  8    Dose1,Dose2; FIELD=9

 Dose1 standard   double   double standard

 Dose2 standard   double   double standard standard   double   double

 Dose2 standard

  9  " Form Dose3 from Dose1, matching by levels (the default)
-10    and then Dose4, matching by ordinal positions of the levels."
 11  FACTOR [NVALUES=4; LEVELS=!(0,1,2,3,4); \
 12    LABELS=!T(none,d1,d2,d3,d4)] Dose3
 13  FACTOR [NVALUES=4; LEVELS=!(20,40,60)] Dose4
 14  EQUATE Dose1; NEW=Dose3
 15  EQUATE [FREPRESENTATION=ordinals] Dose1; NEW=Dose4
 16  PRINT Dose1,Dose3,Dose4

       Dose1       Dose3       Dose4
    standard          d2       20.00
      double          d4       40.00
      double          d4       40.00
    standard          d2       20.00
```

The values of factors that have labels can be copied into texts. In addition, values of texts can be copied into factors, provided all the strings are valid labels for the factor concerned. Factor values can also be copied into variates; the `FREPRESENTATION` option controls whether Genstat uses the levels or the ordinal values.

4.3.2 Comparing sets: the SETRELATE directive

SETRELATE directive

Compares the distinct values contained in two data structures.

Options

FREPRESENTATION = *string token*	How to represent factors in a comparison between two factors (levels, labels, ordinals); default leve
LFACTORIAL = *scalar*	Limit on number of factors or variates in the terms formed from a LEFT formula; default * i.e. none
RFACTORIAL = *scalar*	Limit on number of factors or variates in the terms formed from a RIGHT formula; default * i.e. none
TOLERANCE = *scalar*	Tolerance to use when comparing numerical values; default 10^{-6}
SUBSTITUTE = *string token*	Whether to substitute dummies within LEFT or RIGHT pointers and formulae (yes, no); default no

Parameters

LEFT = *identifiers*	First structures in each comparison
RIGHT = *identifiers*	Second structures in each comparison
CONTAINS = *scalars*	Returns 1 or 0 according to whether or not LEFT contains RIGHT
EQUALS = *scalars*	Returns 1 or 0 according to whether or LEFT and RIGHT contain exactly the same distinct set of items
INCLUDEDIN = *scalars*	Returns 1 or 0 according to whether or not LEFT is included in RIGHT

SETRELATE can compare the distinct values of any numerical structure (scalar, variate, table, matrix, diagonal matrix or symmetric matrix) with another numerical structure or with a factor. It can compare a factor either with another factor, or with a variate or a text. It can compare a text with another text, or two pointers. Finally, it can compare two formulae.

The LEFT and RIGHT parameters specify the structures to compare. The other parameters provide the results of the comparison as scalars containing the values 0 or 1. CONTAINS is set to 1 if the LEFT structure contains every (distinct) value in the RIGHT structure. EQUALS returns 1 if the sets of distinct values in the LEFT and RIGHT structures are identical. INCLUDEDIN equals 1 if the RIGHT structure contains every (distinct) values in the LEFT structure.

When comparing two factors, the FREPRESENTATION option specifies whether to use levels, labels or ordinal values. (The ordinal values are formed representing the levels, in numerical order, by the numbers 1, 2 and so on.) By default levels are used.

When comparing pointers and formulae, the SUBSTITUTE option controls whether any dummies that they contain are substituted by the data stuctures to which they point, before the comparison is made. Note, if any of those data structures is a dummy, it too is replaced, and so on until we reach a data structure that is *not* a dummy. However, if the original dummy (or any of the dummies to which it points) is unset, the original dummy is retained.

The LFACTORIAL and RFACTORIAL options can be used to set a limit on number of factors or variates in the terms formed from a LEFT or RIGHT formula, respectively; by default there are no limits.

Example 4.3.2 compares the examination results of three students. SETRELATE used to find out that Jim and Kim have both passed exams that were not passed (or perhaps not taken) by the other. However, Tim's exam successes are a subset of Kim's.

Example 4.3.2

```
" Examinations passed by three students "
TEXT Jim,Kim,Tim; VALUES=\
     !t(Art,English,French,Geography,History,\
        'Information technology',Maths,Music,Science,Spanish),\
     !t('Business studies',English,French,Geography,History,\
        Maths,Music,'Resistant materials',Science,Spanish),\
     !t('Business studies',English,French,Geography,History,\
        'Resistant materials')

" Compare Jim with Kim: neither set of exams is a subset of the other"
SETRELATE Jim; Kim; CONTAINS=Kim_subset_of_Jim;\
          EQUALS=Jim_same_as_Kim; INCLUDEDIN=Jim_subset_of_Kim
PRINT     [ORIENTATION=across; RLWIDTH=20]\
          Kim_subset_of_Jim,Jim_same_as_Kim,Jim_subset_of_Kim;\
          DECIMALS=0

Kim_subset_of_Jim          0
  Jim_same_as_Kim          0
Jim_subset_of_Kim          0


" but Tim's exams are a subset of Kim's."
SETRELATE Kim; Tim; CONTAINS=Tim_subset_of_Kim;\
          EQUALS=Kim_same_as_Tim; INCLUDEDIN=Kim_subset_of_Tim
PRINT     [ORIENTATION=across; RLWIDTH=20]\
          Tim_subset_of_Kim,Kim_same_as_Tim,Kim_subset_of_Tim;\
          DECIMALS=0

Tim_subset_of_Kim          1
  Kim_same_as_Tim          0
Kim_subset_of_Tim          0
```

SETRELATE also has a TOLERANCE option, which can be used to change the tolerance used internally to compare numbers. The default value 10^{-6} should be suitable, however, unless you are working with very small numbers.

4.3.3 Boolean arithmetic: the **SETCALCULATE** directive

SETCALCULATE directive

Performs Boolean set calculations on the contents of vectors or pointers.

Options

NULL = *scalar*	Returns either 1 or 0 according to whether or not the result is a null (i.e. empty) set
FREPRESENTATION = *string token*	How to represent factors in a calculation that contains only factors (levels, labels); default leve
TOLERANCE = *scalar*	Tolerance to use when comparing numerical values; default 10^{-6}
SUBSTITUTE = *string token*	Whether to substitute dummies within pointers in the expression (yes, no); default no
NOMESSAGE = *string tokens*	Which warning messages to suppress (novalues); default * i.e. none

Parameter

expression	Expression defining the calculation to be performed

The `SETCALCULATE` directive allows you to perform calculations with the sets of distinct values contained in a vector or pointer. The calculation must have a single assignment, setting a pointer, variate, text or factor to the result of a set calculation involving other compatible structures. Calculations on pointers must involve only pointers, those on variates and those on texts can involve factors, while those on factors can involve either variates or texts but not both.

For example, you can form a variate `All` that contains all the distinct values that occur in either of a pair of variates `x` and `y` using the `.OR.` operator

```
SETCALCULATE All = x .OR. y
```

or all the (distinct) values that occur in both of them using the `.AND.` operator

```
SETCALCULATE Both = x .AND. y
```

The available operators are as follows:

`.OR.`	represents the Boolean "or" operation: for example `x.OR.y` produces a vector that contains any item that is in either `x` or `y`
`.AND.`	represents the Boolean "and" operation: for example `x.AND.y` produces a vector that contains any item that is in both `x` and `y`
`.EOR.`	represents "either or": for example `x.EOR.y` produces a vector that contains any item that is in either `x` or `y` but not both of them
`-`	represent "not", for example `x-y` produces a vectors that contains the items that are in `x` but not in `y`
`+`	synonym of `.OR.`
`,`	synonym of `.OR.`
`*`	synonym of `.AND.`

The `NULL` option can save a scalar whose value is 1 if the calculation generates an null set (i.e. one that has no members); otherwise the scalar is set to 0. The `FREPRESENTATION` option determines whether the values of factors are compared using their levels or their labels; by default the levels are used. The `TOLERANCE` option defines the tolerance to be used when comparing numbers. The default value 10^{-6} should be suitable, however, unless the variates contain very small values. If the calculation is operating on pointers, the `SUBSTITUTE` option controls whether or not to replace any dummies that they contain by the structures to which they point. Finally, the `NOMESSAGES` option allows you to suppress the warning message that `SETCALCULATE` produces if one of the data structures in the calculation has no values. Finally, the `NOMESSAGES` option allows you to suppress the warning message that `SETCALCULATE` produces if one of the data structures in the calculation has no values.

Example 4.3.3 uses `SETCALCULATE` to make a further investigation of the examination results in Example 4.3.2.

Example 4.3.3

```
  " All exams passed by any of the students "
  SETCALCULATE All_exams = Jim .OR. Kim .OR. Tim
  PRINT        All_exams; JUSTIFICATION=left

All_exams
Art
English
French
Geography
History
Information technology
Maths
Music
Science
```

```
Spanish
Business studies
Resistant materials

  28
  29  " Exams passed by both Jim and Kim "
  30  SETCALCULATE Jim_and_Kim = Jim .AND. Tim
  31  PRINT        Jim_and_Kim; JUSTIFICATION=left

Jim_and_Kim
English
French
Geography
History

  32
  33  " Exams passed by Kim but not Tim "
  34  SETCALCULATE Kim_not_Tim = Kim - Tim
  35  PRINT        Kim_not_Tim; JUSTIFICATION=left

Kim_not_Tim
Maths
Music
Science
Spanish

  36
  37  " Exams passed by only one student "
  38  SETCALCULATE One_student = (Jim - (Kim .OR. Tim))\
  39                        .OR. (Kim - (Jim .OR. Tim))\
  40                        .OR. (Tim - (Jim .OR. Kim))
  41  PRINT        One_student; JUSTIFICATION=left

One_student
Art
Information technology
```

4.3.4 All subsets of a set of objects: the `SETALLOCATIONS` directive

`SETALLOCATIONS` directive

Runs through all ways of allocating a set of objects to subsets.

Options

`NREQUIRED` = *scalar*	Number of allocations that are required; default 1
`UNIQUE` = *string token*	Whether only unique allocations are to be formed, allowing the reordering of the subsets (`yes`, `no`); default `no`
`NFOUND` = *scalar*	Number of allocations that has been found
`NPOSSIBLE` = *scalar*	Saves the total of allocations that can be formed
`GROUPS` = *factor* or *pointer*	Saves the allocations, in a single factor if `NREQUIRED` = 1, otherwise in a pointer to `NFOUND` factors
`UNITS` = *variate*	Supplies numbers for the objects; if unset, the positive integers 1, 2 ... are used
`START` = *factor*	Previous allocation; if unset the allocations start as a partitioning of the objects in the ordering in the `UNITS` variate

Parameters

`SETSIZE` = *scalars*	Number of objects in each subset

ELEMENTS = *variates* or *pointers*	Saves the objects allocated to each subset, in a single variate if NREQUIRED = 1, otherwise in a pointer to NFOUND variates

The SETALLOCATIONS directive allows you to form all the ways in which a set of objects can be allocated to subsets. For example, suppose we have 4 objects numbered 1 ... 4 to allocate to two subsets of size 2. Then, as Example 4.3.4 shows, there are 6 possible ways of forming the allocations: {1, 2 : 3, 4}, {1, 3 : 2, 4}, {1, 4 : 2, 3}, {2, 3 : 1, 4}, {2, 4 : 1, 3} and {3, 4 : 1, 2}. (The first subsets are in the variates Sub1[1...6] and the second subsets are in the variates Sub1[1...6].) If, however, the ordering of the subsets is unimportant, there are only three. For example {1, 2 : 3, 4} is then the same as {3, 4 : 1, 2}.

Example 4.3.4

```
  2  SCALAR         nreqd
  3  SETALLOCATIONS [NFOUND=Nfound; NREQUIRED=nreqd; GROUPS=Alloc] 2,2;\
  4                 ELEMENTS=Sub1,Sub2
  5  PRINT          Nfound

     Nfound
      6.000

  6  &              Alloc[]

   Alloc[1]    Alloc[2]    Alloc[3]    Alloc[4]    Alloc[5]    Alloc[6]
          1           1           1           2           2           2
          1           2           2           1           1           2
          2           1           2           1           2           1
          2           2           1           2           1           1

  7  &              Sub1[1...6]; DECIMALS=0 & Sub2[1...6]; DECIMALS=0

    Sub1[1]     Sub1[2]     Sub1[3]     Sub1[4]     Sub1[5]     Sub1[6]
          1           1           1           2           2           3
          2           3           4           3           4           4


    Sub2[1]     Sub2[2]     Sub2[3]     Sub2[4]     Sub2[5]     Sub2[6]
          3           2           2           1           1           1
          4           4           3           4           3           2
```

The sizes of the subsets are specified by the SETSIZE parameter, and the UNIQUE option can be set to yes to indicate that their ordering is unimportant (so only *unique* allocations are then formed). The NREQUIRED option indicates how many allocations you want to form (default 1). If you set NREQUIRED to a scalar containing a missing value, like the scalar nreqd in line 2 above, SETALLOCATIONS will save as many allocations as it can find.

You can use the NPOSSIBLE option to find out how many allocations are possible. The NFOUND option is useful if you request more allocations than are possible – it indicates how many allocations SETALLOCATIONS has actually been able to find.

The GROUPS option saves the allocations in factors (each with a level for each subset). If NREQUIRED = 1, it saves a single factor. Alternatively, if NREQUIRED is greater than one, it saves a pointer containing NFOUND factors.

As an alternative, the ELEMENTS parameter allows you to save the allocations in variates. For example

```
    SETALLOCATIONS 2,2; ELEMENTS=Sub1, Sub2
```

saves the first subset in variate Sub1 and the second in variate Sub2. Just one allocation has been formed here as the NREQUIRED option has default 1. If several allocations are formed, ELEMENTS saves them in pointers to variates. For example

```
SETALLOCATIONS [NREQUIRED=3] 2,2; ELEMENTS=Sub1, Sub2
```

saves three allocations: (`Sub1[1] : Sub2[1]`), (`Sub1[2] : Sub2[2]`), and (`Sub1[3] : Sub2[3]`). By default, the variates will contain the positive integers 1, 2 upwards, but you can supply a variate containing others using the `UNITS` option.

By default, the first allocation is a partitioning of the objects in the same ordering as in the `UNITS` variate (or numerical order if `UNITS` is not set). However, if you want to run through the allocations in order, you can save the current allocation using the `GROUPS` option, and then use this as the setting of the `START` option to get the next one. For example, the following program runs through all the allocations of seven objects into subsets of size 2, 3 and 2.

```
SETALLOCATIONS   [NPOSSIBLE=Nposs; GROUPS=Alloc] 2,3,2
CALCULATE        Ntimes = Nposs - 1
FOR [INDEX=i; NTIMES=Ntimes]
  DUPLICATE      Alloc; NEWSTRUCTURE=Start
  SETALLOCATIONS [NREQUIRED=1; GROUPS=Alloc; START=Start]\
                 2,3,2
  " use allocation Alloc "
ENDFOR
```

4.3.5 Locations of a value in a data structure: the `GETALLOCATIONS` directive

`GETLOCATIONS` directive

Finds locations of an identifier within a pointer, or a string within a factor or text, or a number within any numerical data structure.

Options

`CASE` = *string token*	Whether to treat the case of letters (small or capital) as significant when searching for a string (`significant`, `ignored`); default `sign`
`TOLERANCE` = *scalar*	Tolerance for comparing numbers
`SUBSTITUTE` = *string token*	Whether to substitute dummies within pointers in `DATA` or `FIND` (`yes`, `no`); default `no`

Parameters

`DATA` = *identifiers*	Variates, scalars, matrices, tables, factors, texts or pointers to be searched
`FIND` = *scalars*, *texts* or *pointers*	Numbers, strings or identifiers to be located in `DATA`
`NLOCATIONS` = *scalars*	Saves the number of times that `FIND` occurs in `DATA`
`LOCATIONS` = *variates* or *pointers*	Saves the locations where `FIND` occurs as one of the values in `DATA`, in a variate if `DATA` is a one-dimensional data structure like a variate or text, or in a pointer containing a variate for each dimension if `DATA` is a multi-dimensional data structure like a matrix or table
`CLASSIFICATION` = *pointers*	Saves the classifying factors for a `DATA` table, in the same order as the corresponding variates in the `LOCATIONS` and `LEVELS` pointers
`LEVELS` = *pointers*	Saves the levels of the classifying factors where `FIND` occurs as one of the values of a `DATA` table, the information is saved in a pointer containing a variate for each factor

The GETLOCATIONS directive finds where a particular item occurs as one of the values stored by a Genstat data structure. So, for example, it can locate the lines within a text structure that are equal to a particular string, or it can locate the rows and columns of a matrix that hold a particular number, or it can locate the numbers of the suffixes where a pointer contains a particular identifier.

The item to find is specified by the FIND parameter, as a scalar (for a number), or a single-valued text (for a string of characters), or a single-valued pointer (for the identifier of a data structure). The data structure to search is supplied by the DATA parameter.

If the FIND pointer contains a dummy, GETLOCATIONS usually looks to see that dummy is contained in the DATA pointer. Alternatively, if you set option SUBSTITUTE=yes and the FIND pointer contains a dummy, it is replaced by the data structure to which it points. Then if that data structure is a dummy, it too is replaced, amd so on until we reach a data structure that is *not* a dummy. However, if the original dummy (or any of the dummies to which it points) is unset, the original dummy is retained. The same substitution is done on any dummies in the DATA pointer. So, when SUBSTITUTE=yes, GETLOCATIONS matches the structures to which the dummies (eventually) point, rather than the dummies themselves.

The number of times that FIND occurs in DATA can be saved, in a scalar, by the NLOCATIONS parameter. The locations where FIND occurs can be saved by the LOCATIONS parameter. These are saved in a variate if DATA is a one-dimensional structure i.e. a scalar, variate, text, factor or pointer. If DATA is a rectangular, diagonal or symmetric matrix, they are saved in a pointer containing two variates. The first saves the row locations, and the second saves the column locations. If DATA is a table, LOCATIONS saves a pointer with a variate for each factor classifying the table. Each variate stores the locations within the dimension classified by a particular factor. So, for example, with a two-way DATA table the first variate stores the row numbers, and the second variate stores the column numbers. The CLASSIFICATION parameter can save the factors in the same order as the LOCATIONS variates, in case you are unsure of which factor corresponds to each dimension. The LEVELS parameter provides an alternative to LOCATIONS for tables, storing the factor levels corresponding to the positions in each dimension, rather than the numbers of e.g. the rows or columns. If a DATA table has margins and the number to FIND occurs in one of them, a missing value will be stored for the corresponding location or level.

Example 4.3.5 uses GETLOCATIONS to find where the number 2 occurs within the variate X.

Example 4.3.5

```
   2  VARIATE [VALUES=1,3,2,5,7,2,1,4,5,2,8,5,6,1,4,8,2] X
   3  GETLOCATIONS X; FIND=2; LOCATIONS=Loc
   4  PRINT Loc

         Loc
           3
           6
          10
          17
```

4.4 Operations on vectors

The directives and procedures described below can be used with any of the vector structures that Genstat supports: variates, factors or texts. More specific facilities are described in 4.5 (variates), 4.6 (factors) and 4.7 (texts).

The RESTRICT directive (4.4.1) allows you to indicate that future statements should operate only on a subset of the units of the specified vectors. (The precise way in which RESTRICT affects the operation of other directives is described in the chapters that are devoted to these directives.) This is a convenient way of saving space when you wish to examine successive subsets, as there is no need to create a copy of the subset; the vectors themselves are unchanged and merely have some *restriction* associated with them. The restriction can be changed or cancelled at any time by specifying RESTRICT again. This would, for example enable you to analyse a data variate, taking one subset at a time while building up full variates of residuals and fitted values that contain the information from all the subset analyses.

Alternatively, if you wish to look at a single subset, the SUBSET procedure (4.4.2) allows you to create a set of vectors which contain only a subset of the original vectors.

The SORT directive allows you to reorder the units of vectors according to the values of one or more index vectors (4.4.3). You can use the RANDOMIZE directive, described in 2:4.11.1, to put units into random order.

The APPEND procedure allows you to append values from a set of vectors (4.4.4) into another vector of a compatible type. The STACK procedure (4.4.5) performs a similar operation but appends complete data sets together rather than individual vectors, while the converse operation of splitting the vectors back into their original components is provided by the UNSTACK procedure (4.4.6). Finally, section 4.4.7 describes another way of combining data sets, by adding new columns instead of new units, which can be done using the JOIN procedure.

4.4.1 Applying a restriction to the units of a vector: the RESTRICT directive

RESTRICT directive

Defines a restricted set of units of vectors for subsequent statements.

No options

Parameters

VECTOR = *vectors*	Vectors to be restricted
CONDITION = *expression*	Logical expression defining the restriction for each vector; a zero (false) value indicates that the unit concerned is not in the set
SAVESET = *variates*	List of the units in each restricted set
NULL = *scalars*	Indicator for each restricted set, set to 1 or 0 according to whether or not it contains no units

The RESTRICT directive defines a *restriction* on the units of a vector, so that future operations will involve only a subset of the units. The directives that take account of RESTRICT are listed at the end of this subsection.

The VECTOR parameter specifies the vector or vectors that are to be restricted. These can be variates, factors or texts, but all the vectors listed must be of the same length.

The CONDITION parameter specifies a logical expression which indicates which units of the vectors are in the defined subset. For example,

```
VARIATE [VALUES=1,2,3,2,3,4,3,4,5] V
RESTRICT V; CONDITION=V.EQ.2
```

restricts the vector V to those units with the value 2. Genstat evaluates the expression to generate internally a variate of zeroes and ones, of the same length as the vectors being restricted. A zero value indicates that the corresponding unit is to be excluded. The logical expression can involve any vector of the same length as the vector to be restricted. For example, to restrict variate V and text T to the units with levels 1 or 2 or 4 of factor F, you could use the statement

```
RESTRICT V,T; CONDITION=(F.LE.2).OR.(F.EQ.4)
```

When using a text to define a restriction, remember that you cannot use logical operators like .EQ. and .NE. Instead you should use operators .IN., .NI., .EQS. and .NES. (4.1.2):

```
TEXT [VALUES=London,Madrid,Nairobi,Ottawa,Paris,\
  Quito,Rome] City
& [VALUES=London,Madrid,Paris,Rome] Europe
RESTRICT City; CONDITION=City.IN.Europe
```

restricts the text City to lines 1, 2, 5 and 7 only.

Of course, the expression may just contain a single variate of the of the same length as the vectors to be restricted. Again a zero indicates that the corresponding unit in the vector to be restricted is excluded, while any non-zero entry causes inclusion. Thus the restriction above on the text `City` could also be specified by

```
   RESTRICT T; CONDITION=!(1,1,0,0,1,0,1)
```

The same effect can be achieved by using the `EXPAND` function (4.2.8):

```
   RESTRICT City; CONDITION=EXPAND(!(1,2,5,7))
```

Another function that may be useful is `RESTRICTION`; this allows you to generate a variate of ones and zeros indicating the units to which a vector is currently restricted (4.2.8). It thus provides a very convenient way of transferring a restriction from one vector to another. For example,

```
   RESTRICT Timezone,Distance; CONDITION=RESTRICTION(City)
```

restricts the vectors `Timezone` and `Distance` to the same units as those to which `City` is currently restricted.

Finally, if you omit the `CONDITION` parameter, this removes any restrictions on the vectors are removed. For example

```
   RESTRICT City,Timezone,Distance
```

removes any restrictions that have been set on `City`, `Timezone` and `Distance`.

Note that if the vectors used in the `CONDITION` expression are themselves restricted these restrictions will remain in force during the current calculation of the condition. A danger here, therefore, is that you may accidentally end up restricting out all the elements of a vector by using `RESTRICT` repeatedly. The safest way to avoid this is to remove the restrictions on any vectors to be used in the `CONDITION` expression before you use them to restrict vectors in some different way.

The `SAVESET` parameter can be used to save the numbers of the units that are in the restricted set. These are saved in a variate with one value for each unit retained by the restriction. Thus, if the example above with variate V were to become

```
   VARIATE [VALUES=1,2,3,2,3,4,3,4,5] V
   RESTRICT V; CONDITION=V.EQ.2; SAVESET=S
```

`S` would be created as a variate of length 2, with values 2 and 4.

The `NULL` parameter can specify a list of scalars, one for each vector in the `VECTOR` list, that will be set to one if its restricted set contains no units; otherwise it is set to zero. Also, when `NULL` set, `RESTRICT` suppresses the warnings that it normally gives if a restricted set is null.

Not all directives take account of `RESTRICT`. For those that do, usually only one vector in the list of parameters has to be restricted for the directive to treat them all as being restricted in the same way. A fault is reported if any vectors in such a list are restricted in different ways.

A general guideline is that `RESTRICT` is obeyed by all directives that operate on vectors except in those statements where explicit identification of elements is possible: for example `EQUATE` (4.3.1) and `READ` (3.1.2), and when qualified identifiers or `ELEMENT` functions are used in `CALCULATE` (4.1.6 and 4.2.8). However, this guideline is not always followed. There is a section on `RESTRICT` in the *Reference Manual* for each command where this is relevant; see Part 2 (for directives) or Part 3 (for procedures).

4.4.2 Forming a subset of the units in a vector: the SUBSET procedure

SUBSET procedure

Forms vectors containing subsets of the values in other vectors (R.W. Payne).

Options

CONDITION = *expression*	Logical expression to define which units are to be included; no default – this option must be set
SETLEVELS = *string token*	Whether to reform the levels (and labels) of factors to exclude those that do not occur in the subset (yes, no); default no
NULL = *scalar*	Indicator set to 1 or 0 according to whether or not the subset contains no units

Parameters

OLDVECTOR = *vectors*	Vector from which the subset is to be formed
NEWVECTOR = *vectors*	Vector to store the subsets if none is specified, the OLDVECTOR is redefined to store the subset

Procedure SUBSET forms vectors containing subsets of the values in other vectors. The subset is defined by a logical condition which must be specified by the CONDITION option; units with non-zero and non-missing values (true) for the condition are included in the subset, others are omitted. Subsets can be formed for factors, texts and variates. Relevant attributes will also be transferred across to the new structures but, if the subset excludes some of the levels of a factor, a new reduced set of levels (and labels) can be requested by setting option SETLEVELS=yes.

The NULL option can specify a scalar that will be set to one if the subset contains no units; otherwise it is set to zero. Also, when NULL set, SUBSET suppresses the fault that it normally gives if the subset is empty.

The original vectors are specified by the OLDVECTOR parameter and identifiers for the vectors to contain the subsets are specified by the NEWVECTOR parameter. If NEWVECTOR is not set, the OLDVECTOR are redefined to store the subsets instead of their original values.

Below we show an example.

Example 4.4.2

```
VARIATE [VALUES=101...126] X
TEXT [VALUES=a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z] T
FACTOR [LEVELS=26; VALUES=1...26] F
SUBSET [CONDITION=X<111] OLDVECTOR=X,T,F; NEWVECTOR=Xs,Ts,Fs
PRINT  Xs,Ts,Fs; FIELD=12

          Xs          Ts          Fs
       101.0           a           1
       102.0           b           2
       103.0           c           3
       104.0           d           4
       105.0           e           5
       106.0           f           6
       107.0           g           7
       108.0           h           8
       109.0           i           9
       110.0           j          10
```

4.4.3 Sorting vectors into numerical or alphabetical order: the SORT directive

SORT directive

Sorts units of vectors according to an index vector.

Options

INDEX = *vectors*	Variates, texts or factors whose values are to define the ordering; default is to use the first vector in the OLDVECTOR list
DIRECTION = *string token*	Order in which to sort (ascending, descending); default asce
DECIMALS = *scalar*	Number of decimal places to which to round before sorting numbers; default * i.e. no rounding

Parameters

OLDVECTOR = *vectors* or *pointers*	Factors, pointers, texts or variates whose values are to be sorted
NEWVECTOR = *vectors* or *pointers*	Structure to receive each set of sorted values; if any are omitted, the values are placed in the corresponding OLDVECTOR

The SORT directive allows you to reorder the units of a list of vectors or pointers according to one or more "index" vectors. These can be specified explicitly using the INDEX option. If you omit the INDEX option, Genstat uses the first vector in the OLDVECTOR list. The DECIMALS option allows you to define the number of decimal places that are taken into account for an index variate: for example DECIMALS=0 would round each value to the nearest integer. If you do not set this, there is no rounding. The DIRECTION option controls whether the ordering is into ascending or descending order; by default DIRECTION=ascending. If there are ties in the index vector(s), SORT keeps the units concerned in their original order.

The vectors or pointers whose values are to be sorted are listed by the OLDVECTOR parameter. The units of each structure are permuted in exactly the same way, into an ordering determined from the index vectors.

In Example 4.4.3a, the units of the variates Age and Income, the text Name, and the factor Sex are sorted to put the names into alphabetical order.

Example 4.4.3a

```
   2  VARIATE [VALUES=18,50,24,49,61,29,32,42,36,40] Age
   3  & [VALUES=3000,17500,5000,20000,7000,4500, \
   4    12000,18000,15500,17500] Income
   5  TEXT [VALUES=Clarke,Innes,Adams,Jones,Day, \
   6    Grey,Edwards,Baker,Hill,Foster]   Name
   7  FACTOR [LABELS=!T(male,female); VALUES=2,1,1,1,2,2,1,1,2,1] Sex
   8  PRINT Age,Income,Name,Sex; FIELD=12

         Age      Income        Name         Sex
       18.00        3000      Clarke      female
       50.00       17500       Innes        male
       24.00        5000       Adams        male
       49.00       20000       Jones        male
       61.00        7000         Day      female
       29.00        4500        Grey      female
       32.00       12000     Edwards        male
       42.00       18000       Baker        male
       36.00       15500        Hill      female
       40.00       17500      Foster        male
```

```
  9   SORT [INDEX=Name] Age,Income,Name,Sex
 10   PRINT Age,Income,Name,Sex; FIELD=12

          Age      Income        Name         Sex
        24.00        5000       Adams        male
        42.00       18000       Baker        male
        18.00        3000      Clarke      female
        61.00        7000         Day      female
        32.00       12000     Edwards        male
        40.00       17500      Foster        male
        29.00        4500        Grey      female
        36.00       15500        Hill      female
        50.00       17500       Innes        male
        49.00       20000       Jones        male
```

Here the index vector `Name` is also one of the vectors being sorted; indeed if you list it first, then you can omit the `INDEX` option:

```
SORT Name,Age,Income,Sex
```

However it need not be among the vectors sorted. Moreover, you can specify new vectors to contain the sorted values, and thus keep the unsorted values in the original vectors. For example

```
SORT [INDEX=Name] Age,Income,Name,Sex; NEWVECTOR=A,*,N,S
```

would place the sorted values of `Age`, `Name` and `Sex` into `A`, `N` and `S`; as there is a null entry (`*`) corresponding to `Income` in the `NEWVECTOR` list, the sorted incomes would replace the original values of `Income`. Any undeclared vector in the `NEWVECTOR` list is declared implicitly to match the corresponding `OLDVECTOR`.

We now sort the units into order of descending age.

Example 4.4.3b

```
 11   SORT [DIRECTION=descending] Age,Income,Name,Sex
 12   PRINT Age,Income,Name,Sex; FIELD=12

          Age      Income        Name         Sex
        61.00        7000         Day      female
        50.00       17500       Innes        male
        49.00       20000       Jones        male
        42.00       18000       Baker        male
        40.00       17500      Foster        male
        36.00       15500        Hill      female
        32.00       12000     Edwards        male
        29.00        4500        Grey      female
        24.00        5000       Adams        male
        18.00        3000      Clarke      female
```

Here there is a variate as index vector. The `DIRECTION` option can also apply to textual index vectors, when ascending order is interpreted as alphabetical order. The default for `DIRECTION` is to sort into ascending order.

Finally, to illustrate the use of more than one index vector, we sort into increasing order of incomes, taking alphabetic ordering of the names where two people earn the same amount.

Example 4.4.3c

```
 13   SORT [INDEX=Income,Name] Age,Income,Name,Sex
 14   PRINT Age,Income,Name,Sex; FIELD=12

          Age      Income        Name         Sex
        18.00        3000      Clarke      female
        29.00        4500        Grey      female
        24.00        5000       Adams        male
        61.00        7000         Day      female
```

```
     32.00       12000     Edwards         male
     36.00       15500        Hill       female
     40.00       17500      Foster         male
     50.00       17500       Innes         male
     42.00       18000       Baker         male
     49.00       20000       Jones         male
```

4.4.4 Appending values of vectors: the **APPEND** procedure

APPEND procedure

Appends a list of vectors of a compatible type (R.W. Payne).

Options

NEWVECTOR = *variate*, *factor* or *text*	Vector to store the appended values; by default uses the first vector of the OLDVECTOR list
FREPRESENTATION = *string token*	How to match the values of old factors (levels, labels, ordinals, renumbered); default leve
GROUPS = *factor*	Factor to represent the OLDVECTOR to which each unit originally belonged

Parameter

OLDVECTOR = *variates*, *factors*, *texts* or *scalars*	Values to be appended

APPEND provides a convenient way of taking the values from several variates, factors, scalars or texts and appending (i.e. copying) them into a single variate, factor or text. The variates, factors scalars and texts whose values are to be appended are specified by the OLDVECTOR parameter, and the NEWVECTOR option supplies the variate, factor or text to store the appended values. If NEWVECTOR is omitted, the values are placed into the first OLDVECTOR (but it must not be a scalar). Also, the type of the NEWVECTOR is taken from the first OLDVECTOR, if it has not already been defined.

The NEWVECTOR will contain all the values of the first OLDVECTOR, then all those from the second, and so on. The old vectors can thus contain different numbers of values, but they must be of compatible types. Texts can receive values from any type of OLDVECTOR, with the values of variates, factors scalars first being formed into texts using the TXCONSTRUCT directive (4.7.2). However, variates cannot receive values from texts. Factors can receive values from any type, subject to the setting of the FREPRESENTATION option, described below. Variates, texts and scalars are first formed into factors, using the GROUPS directive (4.6.1), and the values are then transferred into the new factor. A factor formed from a text will therefore have both levels and labels, but those formed from variates or scalars will have only levels.

The FREPRESENTATION option indicates how the levels of factors are to be matched amongst the old factors. If this is set to labels and the levels of the old factors are compatible (that is if each label corresponds to the same level in all the old factors), then the level definitions are also transferred to the new factor, as shown in line 14 of Example 4.4.4.; if not, the levels are defined to be the default values 1, 2... and a warning is printed. Similarly, with the default setting FREPRESENTATION=levels, the labels are retained if they are compatible, but no warning is printed if they are not. For FREPRESENTATION=ordinals, the levels of all the factors are taken as the ordinal values 1, 2... (and no labels are defined). Finally, the renumbered setting assumes that the old factors all have independent sets of levels, and renumbers these from one upwards for the first factor, from number of levels of the first factor plus one upwards for the second factor, and so on; the new factor will thus have a different level for every level of the original factors.

The `GROUPS` option allows a factor to be formed indicating the `OLDVECTOR` to which each unit of the appended vector originally belonged. The levels are labelled by the identifier of the corresponding `OLDVECTOR`, as shown in lines 8 and 9 of Example 4.4.4. This factor could then be used in the `CONDITION` option of the `SUBSET` procedure (4.4.2) subsequently to recover the original values.

Example 4.4.4

```
   2  VARIATE [VALUES=1...3] V1
   3  &       [VALUES=11,12] V2
   4  &       [VALUES=31] V3
   5  TEXT    [VALUES=a,b,c] T1
   6  &       [VALUES=k,l] T2
   7  &       [VALUES=z] T3
   8  APPEND  [NEWVECTOR=Newvar; GROUPS=Vname] V1,V2,V3
   9  APPEND  [NEWVECTOR=Newtext; GROUPS=Tname] T1,T2,T3
  10  PRINT   Newvar,Vname,Newtext,Tname; FIELD=12

      Newvar        Vname     Newtext       Tname
        1.00           V1           a          T1
        2.00           V1           b          T1
        3.00           V1           c          T1
       11.00           V2           k          T2
       12.00           V2           l          T2
       31.00           V3           z          T3

  11  FACTOR  [LEVELS=3; LABELS=!t(a,b,c); VALUES=3,2,1] F1
  12  FACTOR  [LEVELS=!(11,12); LABELS=!t(d,e); VALUES=12,11] F2
  13  FACTOR  [LEVELS=!(21...23); LABELS=!t(f,g,h); VALUES=21] F3
  14  APPEND  [NEWVECTOR=Newfac; FREPRESENTATION=labels] F1,F2,F3
  15  PRINT   2(Newfac); FREPRESENTATION=labels,levels; FIELD=12

      Newfac       Newfac
           c        3.000
           b        2.000
           a        1.000
           e       12.000
           d       11.000
           f       21.000
```

4.4.5 Combining data sets: the **`STACK`** procedure

`STACK` procedure

Combines several data sets by "stacking" the corresponding vectors (R.W. Payne).

Option

`DATASET` = *factor*	Factor to indicate the data set to which each unit originally belonged

Parameters

`STACKEDVECTOR` = *variates*, *factors* or *texts*	New vectors combining the corresponding members of the data sets specified by parameter `V1`, or parameters `V1`-`V100`
`V1` = *pointers*, *variates*, *factors*, *texts* or *scalars*	Pointers defining (all) the components to be stacked into each `STACKEDVECTOR`, or contents of the first data set
`V2` - `V100` = *variates*, *factors*, *texts* or *scalars*	

	Data sets 2 - 100
FREPRESENTATION = *string token*	How to match the values of factors (levels, labels, ordinals, renumbered); default leve

STACK allows you to combine vectors (variates, factors or texts) from several data sets into a single data set. Each vector in the new data set is formed by "stacking" the corresponding vectors from the original data sets. So, the new vector first has all the units from the first data set, then those from the second data set, and so on. Note though, that any restrictions on the vectors (applied by RESTRICT: 4.4.1) are ignored.

The identifiers of the new vectors are specified by the first parameter, STACKEDVECTOR. The original vectors of up to 100 data sets can be specified one data set at a time using the subsequent parameters: V1, V2, ... V100. Alternatively, V1 can specify a list of pointers, each one containing all the vectors that are to be stacked together to form the equivalent STACKEDVECTOR (thus allowing vectors from more than 100 data sets to be specified). So, these two statements would be equivalent

```
STACK [DATASET=Month] Rainfall,Temperature;\
  V1=MarchRain,MarchTemp; V2=AprilRain,AprilTemp
```

and

```
STACK [DATASET=Month] Rainfall,Temperature;\
  V1=!p(MarchRain,AprilRain),!p(MarchTemp,AprilTemp)
```

The vectors in each data set must generally all be of the same length. The exception is that you can specify a scalar instead of a variate of identical values (the number of values is then deduced from the lengths of the corresponding vectors of the other data sets). Likewise you can specify a single-valued text instead of a text with duplicates of that value, and either a scalar or a single-valued text instead of a factor with the same level or label duplicated throughout.

The FREPRESENTATION option indicates how the levels are to be matched amongst factors. If this is set to labels and the levels of the original factors are compatible (that is if each label corresponds to the same level in all the original factors), then the level definitions are transferred to the new factor; if not, the levels are defined to be the default values 1, 2... and a warning is printed by the APPEND procedure which is called by STACK. Similarly, with the default setting levels, the labels are retained if they are compatible, but no warning is printed if they are not. For the ordinals setting, the levels of all the factors are taken as the ordinal values 1, 2... (and no labels are defined). Finally, the renumbered setting assumes that the original factors all have independent sets of levels, and renumbers these from one upwards for the first factor, from number of levels of the first factor plus one upwards for the second factor, and so on; the new factor will thus have a different level for every level of the original factors.

The DATASET option allows a factor to be formed indicating the number if the data set to which each unit of the stacked vectors originally belonged. This factor can be used in the DATASET parameter of the UNSTACK procedure subsequently to recover the original vectors (see Example 4.4.6).

The use of STACK is illustrated in Example 4.4.5, which combines some weather observations from March and April, using the DATASET option to form a factor Month to identify when the observations were made. Notice, that we need to provide the labels for Month explicitly by the FACTOR statement in line 21, and that we have used the AFUNITS procedure to form a factor to identify the day of the month as none was supplied in the original data.

Example 4.4.5

```
   2  VARIATE  [NVALUES=31] MarchRain,MarchTemp
   3  READ     MarchRain,MarchTemp
```

```
  Identifier   Minimum      Mean   Maximum    Values   Missing
   MarchRain    0.2000     2.377     4.800        31         0
   MarchTemp     3.000     8.252     12.00        31         0

11  VARIATE  [NVALUES=30] AprilRain,AprilTemp
12  READ     AprilRain,AprilTemp

  Identifier   Minimum      Mean   Maximum    Values   Missing
   AprilRain    0.1000     2.243     4.600        30         0
   AprilTemp     3.600     7.850     12.50        30         0

19  STACK    [DATASET=Month] Rainfall,Temperature;\
20           V1=MarchRain,MarchTemp; V2=AprilRain,AprilTemp
21  FACTOR   [MODIFY=yes; LABELS=!t(March,April)] Month
22  AFUNITS  [BLOCKSTRUCTURE=Month] Day
23  PRINT    Month,Day,Rainfall,Temperature; DECIMALS=2(0),2(1)

     Month         Day    Rainfall Temperature
     March           1         2.7        11.7
     March           2         2.9         7.6
     March           3         1.7         8.0
     March           4         4.2         9.4
     March           5         4.1         3.7
     March           6         0.2        11.4
     March           7         2.5         6.3
     March           8         3.0        11.9
     March           9         0.3         4.5
     March          10         0.6        10.4
     March          11         3.0        11.3
     March          12         0.3         7.3
     March          13         4.2        12.0
     March          14         2.5        10.0
     March          15         3.9         9.0
     March          16         1.2         4.6
     March          17         1.4         5.7
     March          18         3.7        11.8
     March          19         2.9        11.9
     March          20         2.7        10.4
     March          21         0.9         3.0
     March          22         4.7        10.4
     March          23         3.5         6.0
     March          24         3.4         9.2
     March          25         3.3         5.6
     March          26         4.8         7.5
     March          27         1.9        11.7
     March          28         0.9        10.3
     March          29         1.1         3.4
     March          30         0.2         5.7
     March          31         1.0         4.1
     April           1         0.1         6.0
     April           2         3.2        11.0
     April           3         0.6         4.4
     April           4         1.2        11.2
     April           5         1.7         9.1
     April           6         1.4         3.6
     April           7         3.1         8.2
     April           8         3.0         9.7
     April           9         0.9         7.3
     April          10         1.8         3.6
     April          11         3.7        12.5
     April          12         3.9         7.9
     April          13         1.4         9.3
     April          14         4.6        12.2
     April          15         0.8         4.6
     April          16         1.9         5.8
     April          17         3.1         8.8
     April          18         3.6        11.5
     April          19         2.8         8.4
     April          20         2.8        11.7
     April          21         0.3         6.1
     April          22         4.5         4.5
     April          23         4.2         6.2
     April          24         1.6        12.5
```

```
    April         25          2.7          5.8
    April         26          2.7          5.5
    April         27          2.7          9.6
    April         28          1.8          5.8
    April         29          0.4          9.1
    April         30          0.8          3.6
```

4.4.6 The UNSTACK procedure

UNSTACK procedure

Splits vectors into individual vectors according to levels of a factor (R.W. Payne).

Options

DATASET = *factor*	Factor identifying the unstacked data sets
IDSTACKED = *factors*	Factors identifying how the units of the unstacked data sets should be matched
IDUNSTACKED = *factors*	Factors defined to identify these units in the unstacked vectors
MVINCLUDE = *strings*	Which missing values to include (datasets, idstacked); default * i.e. none

Parameters

STACKEDVECTOR = *variates*, *factors* or *texts*	Vectors to be unstacked
DATASETINDEX = *scalars* or *texts*	Level or label of the DATASET factor indicating the group whose units are to be stored in the UNSTACKEDVECTOR; default takes the levels of DATASET one at a time (and then recycling this list to match the other parameters)
UNSTACKEDVECTOR = *variates*, *factors* or *texts*	Unstacked vectors

UNSTACK allows you to split up (or unstack) vectors into individual vectors. The contents of the individual vectors are determined by a factor, specified by the DATASET option. In the simplest case, each original (stacked) vector is split into several new (unstacked) vectors, one for each level of DATASET. The process assumes that the sets are "replicate" sets of data. For example DATASET might correspond to days on which identical sampling schemes were followed. In the most straightforward case, each set contains the same number of observations all stored in an identical order. However, if the observations are in different orders or if some are absent in some of the sets, you can use the IDSTACKED option to specify one or more factors to identify the matching observations within the sets. The IDUNSTACKED option then allows you to save new factors to indicate where the observations are stored in the new (unstacked) vectors. The unstacked vectors are all of the same length, and missing values are inserted for absent observations.

The MVINCLUDE option controls the inclusion of missing values in the unstacked vectors, with the following settings:

idstacked	includes units with missing values for levels of the IDSTACKED factors that do not occur in the data set (otherwise these are omitted), and
datasets	stacked vectors that correspond to data set indexes that do not occur in the data are defined and filled with missing values (otherwise these are left undeclared, and a warning

is given).

By default none of these are included.

There are three parameters. `STACKEDVECTOR` lists the vectors (variates, factors or texts) that are to be split up. `DATASETINDEX` specifies a level of the `DATASET` factor for each member of the `STACKEDVECTOR` list, and `UNSTACKEDVECTOR` specifies a new vector to store the units of the `STACKEDVECTOR` corresponding to that `DATASETINDEX`. So, for example

```
UNSTACK [DATASET=Days] 5(Weight,Height);\
  DATASETINDEX=1,2,3,4,5;\
  UNSTACKEDVECTOR=W1,W2,W3,W4,W5,H1,H2,H3,H4,H5
```

would put the weight measurements made on days 1-5 into `W1`, `W2`, `W3`, `W4` and `W5`, respectively, and the height measurements into `H1`, `H2`, `H3`, `H4` and `H5`. (The construct `5(Weight,Height)` is equivalent to typing `Weight` five times and then `Day` five times, and the `DATASETINDEX` list `1,2,3,4,5` is repeated twice so that it matches the lengths of the other parameter lists.) This method of specification means that you are free to list the vectors and levels in whatever order is most convenient. For example

```
UNSTACK [DATASET=Days] (Weight,Height)5;\
  DATASETINDEX=2(1,2,3,4,5);\
  UNSTACKEDVECTOR=W1,H1,W2,H2,W3,H3,W4,H4,W5,H5
```

lists them in group order rather one stacked vector at a time. If `DATASETINDEX` is not specified, the levels of `DATASET` are taken in order one at a time (and recycled if necessary).

Example 4.4.6 follows Example 4.4.5, and splits the stacked vectors back into their individual components. Notice that the factor `Day` is used to identify the matching observations. This is not vital in this example, as the units are in an identical order within each month, but it would be necessary if the units had been ordered differently in each month.

Example 4.4.6

```
UNSTACK  [DATASET=Month; IDSTACKED=Day; IDUNSTACK=DayOfMonth]\
         2(Rainfall,Temperature); DATASETINDEX='March','April';\
         UNSTACKEDVECTOR=RainMarch,RainApril,TempMarch,TempApril
PRINT    DayOfMonth,RainMarch,RainApril,TempMarch,TempApril;\
         DECIMALS=0,4(1)

DayOfMonth   RainMarch   RainApril   TempMarch   TempApril
         1         2.7         0.1        11.7         6.0
         2         2.9         3.2         7.6        11.0
         3         1.7         0.6         8.0         4.4
         4         4.2         1.2         9.4        11.2
         5         4.1         1.7         3.7         9.1
         6         0.2         1.4        11.4         3.6
         7         2.5         3.1         6.3         8.2
         8         3.0         3.0        11.9         9.7
         9         0.3         0.9         4.5         7.3
        10         0.6         1.8        10.4         3.6
        11         3.0         3.7        11.3        12.5
        12         0.3         3.9         7.3         7.9
        13         4.2         1.4        12.0         9.3
        14         2.5         4.6        10.0        12.2
        15         3.9         0.8         9.0         4.6
        16         1.2         1.9         4.6         5.8
        17         1.4         3.1         5.7         8.8
        18         3.7         3.6        11.8        11.5
        19         2.9         2.8        11.9         8.4
        20         2.7         2.8        10.4        11.7
        21         0.9         0.3         3.0         6.1
        22         4.7         4.5        10.4         4.5
        23         3.5         4.2         6.0         6.2
        24         3.4         1.6         9.2        12.5
        25         3.3         2.7         5.6         5.8
        26         4.8         2.7         7.5         5.5
        27         1.9         2.7        11.7         9.6
        28         0.9         1.8        10.3         5.8
        29          1.1          0.4          3.4          9.1
        30          0.2          0.8          5.7          3.6
        31          1.0            *          4.1            *
```

As with the `STACK` procedure, any restrictions on the vectors (applied by the `RESTRICT` directive) are ignored.

4.4.7 Merging data sets: the `JOIN` procedure

`JOIN` procedure

Joins or merges two sets of vectors together, based on the values of sets of classifying keys (C.F. Johnston & D.B. Baird).

Options

`NINDEX` = *scalar*	Number of index vectors in structures (up to 10); default 1
`METHOD` = *string token*	Type of join (`inner`, `left`, `right`, `full`); default `full`
`REPEATS` = *string token*	How to handle repeats of matches (`combinations`, `single`); default `sing` outputs one row per match
`INCLUDE` = *string token*	How to handle restrictions on the input vectors (`all`, `nonrestricted`); default `all` uses all the data rows
`SORT` = *string token*	Whether `NEWVECTORS` should be sorted on the index vectors (`ascending`, `descending`, `unsorted`); default `unsorted` keeps the same ordering as the input sets

Parameters

`LEFTVECTORS` = *pointer*	Pointer to a list of vectors in left set (keys and variables)
`RIGHTVECTORS` = *pointer*	Pointer to a list of vectors in right set (keys and variables)
`NEWVECTORS` = *pointer*	Pointer to a list of output vectors (keys and variables)

`JOIN` produces a new data set by merging two data sets according to the index (or key) vectors supplied in each data set. `JOIN` supports SQL style joins, as well as merges, as implemented in Genstat for Windows, SAS and SPSS.

The `NINDEX` option specifies the number of index vectors (up to ten). The original data sets are supplied by the `LEFTVECTORS` and `RIGHTVECTORS` parameters. Each of these is a pointer containing the `NINDEX` keys for the data set, followed by any number of extra vectors. The new data set is saved by the `NEWVECTORS` parameter, which is a pointer to `NINDEX` keys followed by the non-index vectors from the two input sets. So, `NEWVECTORS` first contains the combined keys from the left and right sets, then the non-index vectors from the left set, and then the non-index vectors from the right set. `NEWVECTORS` need not be declared in advance, but will be declared automatically if required. The vectors may be variates, factors or texts. Warnings are given if the types of index vectors in each set do not match, although a factor can be matched with a text. Attempting to match a text with a factor or variate will result in a fault.

The `METHOD` option controls the type of join and determines which rows from each input set will be output. `METHOD=inner` outputs only those rows where the keys from both sets match. `METHOD=left` outputs all rows from the `LEFTVECTORS` set and only those rows from `RIGHTVECTORS` where the keys from both sets match. `METHOD=right` outputs all rows from the `RIGHTVECTORS` set and only those rows from `LEFTVECTORS` where the keys from both sets match. `METHOD=full` outputs all rows from both sets. Where keys do not match, missing values are inserted into the non-index vectors from the set without that key value.

The REPEATS option determines what happens when both input sets have repeats of the same matching key values. REPEATS=single outputs one row for each match, so that if there are *m* repeats in LEFTVECTORS and *n* repeats in RIGHTVECTORS, MAX(*m*,*n*) rows will be output. This is the same behaviour as the merge statements of SAS and SPSS and the Merge Spreadsheets menu of Genstat *for Windows*. REPEATS=combinations outputs all combinations of the repeats, giving $m \times n$ rows. This is equivalent to an SQL join and may produce very large output sets.

Setting option INCLUDE=nonrestricted, means that any restrictions on the vectors in each input set will be taken as defining subsets of the rows. (Otherwise restrictions are ignored.) The SORT option allows you to sort the new vectors using the key vectors into either ascending or descending order. By default the vectors are unsorted.

The various methods are illustrated in Example 4.4.7.

Example 4.4.7

```
   2  VARIATE [NVALUES=7] K11,K12,V11
   3  TEXT    [NVALUES=7] T13
   4  READ    K11,K12,V11,V12,T13

    Identifier   Minimum      Mean   Maximum     Values   Missing
           K11     1.000     1.714     3.000          7         0
           K12     1.000     1.571     3.000          7         0
           V11     1.000     4.000     7.000          7         0
           V12     1.900     4.971     7.300          7         0
           T13                                        7         0

  12  VARIATE [NVALUES=5] K21,K22,V21
  13  READ    K21,K22,V21

    Identifier   Minimum      Mean   Maximum     Values   Missing
           K21     1.000     2.200     5.000          5         0
           K22     1.000     1.600     3.000          5         0
           V21     1.000     3.000     5.000          5         0

  19  PRINT K11,K12,V11,V12,T13; FIELDWIDTH=12; DECIMALS=3(0),1,0

         K11         K12         V11         V12         T13
           1           1           1         6.2         Red
           1           1           2         5.7       Green
           1           1           3         4.5        Blue
           1           2           4         7.3         Red
           2           1           5         4.1      Yellow
           3           2           6         5.1        Blue
           3           3           7         1.9       Black

  20  &     K21,K22,V21; DECIMALS=0

         K21         K22         V21
           1           1           1
           1           1           2
           2           1           3
           2           3           4
           5           2           5

  21  JOIN  [NINDEX=2; METHOD=inner; REPEATS=single] \
  22        LEFT=!p(K11,K12,V11,V12,T13); RIGHT=!p(K21,K22,V21); \
  23        NEW=!p(K1,K2,V1,V2,T3,V4)
  24  PRINT K1,K2,V1,V2,T3,V4; FIELDWIDTH=8; DECIMALS=0,0,0,1,0,0

      K1      K2      V1      V2      T3      V4
       1       1       1     6.2     Red       1
       1       1       2     5.7   Green       2
       1       1       3     4.5    Blue       2
       2       1       5     4.1  Yellow       3

  25  JOIN  [NINDEX=2; METHOD=left; REPEATS=single] \
            LEFT=!p(K11,K12,V11,V12,T13); RIGHT=!p(K21,K22,V21); \
            NEW=!p(K1,K2,V1,V2,T3,V4)
      PRINT K1,K2,V1,V2,T3,V4; FIELDWIDTH=8; DECIMALS=0,0,0,1,0,0

      K1      K2      V1      V2      T3      V4
       1       1       1     6.2     Red       1
       1       1       2     5.7   Green       2
       1       1       3     4.5    Blue       2
       1       2       4     7.3     Red       *
       2       1       5     4.1  Yellow       3
       3       2       6     5.1    Blue       *
       3       3       7     1.9   Black       *

      JOIN  [NINDEX=2; METHOD=right; REPEATS=single] \
            LEFT=!p(K11,K12,V11,V12,T13); RIGHT=!p(K21,K22,V21); \
            NEW=!p(K1,K2,V1,V2,T3,V4)
      PRINT K1,K2,V1,V2,T3,V4; FIELDWIDTH=8; DECIMALS=0,0,0,1,0,0

      K1      K2      V1      V2      T3      V4
       1       1       1     6.2     Red       1
       1       1       2     5.7   Green       2
       1       1       3     4.5    Blue       2
       2       1       5     4.1  Yellow       3
       2       3       *       *               4
       5       2       *       *               5

      JOIN  [NINDEX=2; METHOD=full; REPEATS=single; SORT=descending] \
            LEFT=!p(K11,K12,V11,V12,T13); RIGHT=!p(K21,K22,V21); \
            NEW=!p(K1,K2,V1,V2,T3,V4)
      PRINT K1,K2,V1,V2,T3,V4; FIELDWIDTH=8; DECIMALS=0,0,0,1,0,0

      K1      K2      V1      V2      T3      V4
       5       2       *       *               5
       3       3       7     1.9   Black       *
       3       2       6     5.1    Blue       *
       2       3       *       *               4
       2       1       5     4.1  Yellow       3
       1       2       4     7.3     Red       *
       1       1       3     4.5    Blue       2
       1       1       2     5.7   Green       2
       1       1       1     6.2     Red       1

      JOIN  [NINDEX=2; METHOD=full; REPEATS=combinations; SORT=ascending]\
            LEFT=!p(K11,K12,V11,V12,T13); RIGHT=!p(K21,K22,V21); \
            NEW=!p(K1,K2,V1,V2,T3,V4)
      PRINT K1,K2,V1,V2,T3,V4; FIELDWIDTH=8; DECIMALS=0,0,0,1,0,0

      K1      K2      V1      V2      T3      V4
       1       1       1     6.2     Red       1
       1       1       1     6.2     Red       2
       1       1       2     5.7   Green       1
       1       1       2     5.7   Green       2
       1       1       3     4.5    Blue       1
       1       1       3     4.5    Blue       2
       1       2       4     7.3     Red       *
       2       1       5     4.1  Yellow       3
       2       3       *       *               4
       3       2       6     5.1    Blue       *
       3       3       7     1.9   Black       *
       5       2       *       *               5
```

4.5 Operations on variates

Most of the facilities described earlier in this chapter can be used with variates: CALCULATE to perform calculations on their values (4.1.2), RESTRICT to operate only on a subset of their units (4.4.1), SORT to reorder their units (4.4.3), and EQUATE to transfer values between variates and other numerical structures (4.3.1). This section describes directives that only produce variates. The INTERPOLATE directive (4.5.1) allows you to interpolate values at intermediate points of an observed sequence, and MONOTONIC (4.5.2) performs monotonic regressions. The

TX2VARIATE forms a variate from a text, assuming that each of its lines contains a single number or a date (4.5.3).

Procedures for operating on variates include the following: DAYLENGTH to calculate daylengths at a given period of the year HEATUNITS to calculate accumulated heat units of a temperature dependent process; ORTHPOLYNOMIAL to calculate orthogonal polynomials from the values in a variate; SPLINE to form basis functions for M-, B- or I- splines; STANDARDIZE to standardize variates to have mean zero and variance one; VINTERPOLATE to perform linear & inverse linear interpolation between variates; and VTABLE to form a variate and set of classifying factors from a table.

4.5.1 Interpolation

INTERPOLATE directive

Interpolates values at intermediate points.

Options

CURVE = *string token*	Type of curve to be fitted to calculate the interpolated value (linear, cubic); default line
METHOD = *string token*	Type of interpolation required (interval, value, missing): for METHOD=valu, values are interpolated for each point in the NEWINTERVAL variate and stored in the NEWVALUE variate; for METHOD=inte, points are estimated in the NEWINTERVAL variate for the observations in the NEWVALUE variate; while for METHOD=miss, the NEWVALUE and NEWINTERVAL lists are irrelevant, INTERPOLATE now interpolates for missing values in the OLDVALUE and OLDINTERVAL variates (except those missing in both variates); default inte

Parameters

OLDVALUES = *variates*	Observations from which interpolation is to be done
NEWVALUES = *variates*	Results of each interpolation
OLDINTERVALS = *variates*	Points at which each set of OLDVALUES was observed
NEWINTERVALS = *variates*	Points for each set of NEWVALUES

If you have a set of pairs of observations (x, y), you can use interpolation to estimate either a value y for a value x that need not be in the set, or a value x for a value y that likewise need not be in the set. The simplest way to interpolate is by joining successive pairs of observations by straight lines and reading off the appropriate values in between: then the two cases are called *linear interpolation* (obtaining y from x) and *inverse linear interpolation* (obtaining x from y). Genstat can alternatively join the points by cubic functions instead of straight lines. Genstat uses the term *values* to describe the set of y-values and *intervals* for the set of x-values, no matter whether you are doing direct or inverse interpolation.

Genstat does the interpolation for each parallel set of variates in the parameter lists. Each variate in the OLDINTERVALS list specifies the x-values of a set of observed points; the corresponding variate in the OLDVALUES list specifies the corresponding y-values. The variates in the NEWINTERVALS and NEWVALUES lists are for the x-values and y-values of the interpolated points.

If you set METHOD=value, Genstat does ordinary interpolation, and you use the NEWINTERVALS variate to specify the x-values for which you require interpolated y-values.

Genstat calculates the *y*-values and stores them in the corresponding NEWVALUES variate; this variate will be declared implicitly if you have not declared it already.

For the interpolation to take place, the *x*-values must be in either monotonically increasing or decreasing order; thus, if necessary, Genstat takes a copy of the *x*-values and *y*-values and sorts these (in parallel) to put the *x*-values into ascending order.

In Example 4.5.1a, wheat plants have been sampled on five occasions and their growth stage (Zadoks) assessed. The program interpolates values, which it stores in variate Nzad, to estimate the growth stage that the plant has reached after 50, 100 and 150 days.

Example 4.5.1a

```
VARIATE [NVALUES=6] Zadoks,Days; \
  VALUES=!(0,15,23,35,65,95),!(0,50,84,119,147,182)
& [NVALUES=3] Nzadoks,Ndays; VALUES=!(25,50,75),!(50,100,150)
INTERPOLATE [METHOD=value] Zadoks; NEWVALUES=Nzad; \
  OLDINTERVALS=Days; NEWINTERVALS=Ndays
PRINT Ndays,Nzad; FIELDWIDTH=8; DECIMALS=2

   Ndays      Nzad
   50.00     15.00
  100.00     28.49
  150.00     67.57
```

Similarly, if you set METHOD=interval, Genstat does inverse interpolation. You must then specify the *y*-values in the NEWVALUES variate. Genstat calculates the *x*-values and stores them in the corresponding NEWINTERVALS variate, which will be declared implicitly if necessary. Again the *x*-values must be in monotonically increasing or decreasing order, and Genstat will produce a sorted copy if necessary. Inverse interpolation is the default.

Example 4.5.1b uses the same data as above, but does inverse linear interpolation to estimate how long after planting we have to wait for the plant to reach growth stages 25, 50 and 75 Zadoks.

Example 4.5.1b

```
INTERPOLATE [METHOD=interval] Zadoks; NEWVALUES=Nzadoks; \
  OLDINTERVALS=Days; NEWINTERVALS=Nd
PRINT Nzadoks,Nd; FIELDWIDTH=8; DECIMALS=2

 Nzadoks        Nd
   25.00     89.83
   50.00    133.00
   75.00    158.67
```

If you set METHOD=missing, Genstat ignores the NEWVALUES and NEWINTERVALS parameters; it estimates values for *x* or *y* when the other is missing, placing the results in the previously missing position of the OLDVALUES or the OLDINTERVALS variates. Ordinary interpolation is used when the missing value is in *y*, and inverse interpolation when it is in *x*. If both the *x*-value and the *y*-value are missing for a particular unit, no values can be interpolated for it, and it remains missing. To do linear interpolation requires that both the *x*-value and the *y*-value should be non-missing for the point on each side of the unit with the missing value. For cubic interpolation, there must be two non-missing points on each side of the unit. In Example 4.5.1c the missing value in Yval at unit 2 is replaced with the interpolated value 2.85, while the one at unit 4 remains missing because the *x*-value is missing there too. The missing value at unit 9 of Xint is replaced by 5.96, while the one at unit 4 again stays missing. Notice also that Genstat ignores the NEWINTERVALS setting Xnewint.

Example 4.5.1c

```
   2  VARIATE [NVALUES=9] Yval,Xint; \
   3    VALUES=!(2.5,*,3.2,*,4.3,4.8,7.2,7.3,8.7),!(1,2,3,*,4,5,*,6,7)
   4  PRINT Xint,Yval; FIELDWIDTH=8; DECIMALS=2

    Xint    Yval
    1.00    2.50
    2.00       *
    3.00    3.20
       *       *
    4.00    4.30
    5.00    4.80
       *    7.20
    6.00    7.30
    7.00    8.70

   5  INTERPOLATE [METHOD=missing] OLDVALUE=Yval; OLDINTERVAL=Xint ;\
   6    NEWINTERVAL=Xnewint
   7  PRINT Xint,Yval; FIELDWIDTH=8; DECIMALS=2

    Xint    Yval
    1.00    2.50
    2.00    2.85
    3.00    3.20
       *       *
    4.00    4.30
    5.00    4.80
    5.96    7.20
    6.00    7.30
    7.00    8.70
```

The `CURVE` option has two settings, `linear` and `cubic`. By default, `CURVE=linear`, and successive pairs of observations are connected by straight-line segments for linear, or inverse-linear, interpolation. For cubic interpolation you set `CURVE=cubic`; there must then be at least four values in each of the `OLDVALUES` and `OLDINTERVALS` variates.

4.5.2 Monotonic regression

`MONOTONIC` directive

Fits an increasing monotonic regression of y on x.

No options

Parameters

`Y` = *variates*	Y-values of the data points
`X` = *variates*	X-values of the data points; default is to assume that the x-values are monotonically increasing
`RESIDUALS` = *variates*	Variate to save the residuals from each fit
`FITTEDVALUES` = *variates*	Variate to save the fitted values from each fit

Monotonic regression plays a key role in non-metric multidimensional scaling, which is available in Genstat via the `MDS` directive (2:6.12). However, it can be useful in its own right, so the method has been made accessible by the `MONOTONIC` directive. A monotonic regression through a set of points is simply the line that best fits the points subject to the constraint that it never decreases: of course the line need not be straight, in fact it rarely will be. If you need a monotonically decreasing line, you can simply subtract all the y-values from their maximum, find the monotonically increasing regression, and then back-transform the data and fitted line, and change the sign of the residuals.

The `MONOTONIC` directive has no options. It has four parameters: `Y` to specify the y-values, `X` for the x-values, `RESIDUALS` to save the residuals, and `FITTEDVALUES` to save the fitted values. The x-values need not be supplied, in which case the directive assumes that the y-values are in increasing order of the x-values. In common with the other regression directives, the variates to save the residuals and fitted values need not be declared in advance.

In Example 4.5.2, `MONOTONIC` is first used with the data in their original order. The fitted values are saved and plotted as a line, with the data. You can see what happens with the coincident x-values of 4; notice also the horizontal fitted lines that occur when the y-values decrease. Once the data are sorted into increasing order of `X`, at line 7, there is no need to specify the `X` parameter when the `MONOTONIC` directive is used at line 8; as shown by the `PRINT` statement at line 9, the fitted values remain the same.

Example 4.5.2

```
 2  VARIATE [VALUES=2,6,4,4, 9,1,12,15,13,18] X
 3  &       [VALUES=1,5,3,6,10,0,11,14,16,18] Y
 4  MONOTONIC Y=Y; X=X; FITTED=Fvals
 5  LPGRAPH [TITLE='Monotonic regression'; NROWS=25; NCOLUMNS=61] \
 6          Fvals,Y; X; METHOD=line,point

                             Monotonic regression
     -+---------+---------+---------+---------+---------+---------+-
20.0 I                                                             I
     I                                                             I
     I                                                            *I
     I                                                         ..' I
     I                                                       .'    I
     I                                           *        .''      I
15.0 I                                           ........'         I
     I                                           '      *          I
     I                                          '                  I
     I                                         .                   I
     I                                         '                   I
     I                                   ....'*                    I
10.0 I                              *''''                          I
     I                            .'                               I
     I                          .'                                 I
     I                        .'                                   I
     I                      .'                                     I
     I             *.......'                                       I
 5.0 I             .      *                                        I
     I             .                                               I
     I             *                                               I
     I          ..'                                                I
     I        .''                                                  I
     I      .*                                                     I
 0.0 I   *'                                                        I
     -+---------+---------+---------+---------+---------+---------+-
    0.0       3.0       6.0       9.0      12.0      15.0      18.0

 7  SORT X,Y
 8  MONOTONIC Y; RESIDUALS=Res; FITTED=Fvals
 9  PRINT X,Y,Fvals,Res

         X           Y       Fvals         Res
     1.000       0.000       0.000      0.0000
     2.000       1.000       1.000      0.0000
     4.000       3.000       3.000      0.0000
     4.000       6.000       5.500      0.5000
     6.000       5.000       5.500     -0.5000
     9.000      10.000      10.000      0.0000
    12.000      11.000      11.000      0.0000
    13.000      16.000      15.000      1.0000
```

```
     15.000       14.000       15.000      -1.0000
     18.000       18.000       18.000       0.0000
```

4.5.3 Converting a text into a variate

TX2VARIATE directive

Converts text structures to variates.

Options

PRINT = *string token*	Controls printed output (conversions) ; default * (i.e. none)
NONNUMERIC = *string token*	How to treat non-numeric values (bestmatch, missing) default miss
YEAR = *scalar*	Year to use when calculating the day within year for the date formats that specify only months and days; default is to assume that this is any year that is not a leap year
REDEFINE = *string token*	Whether to allow a structure in the VARIATE list that has already been declared (e.g. as a text) to be redefined (yes, no); default no

Parameters

TEXT = *texts*	Text structures to convert
VARIATE = *variates*	Variate for each text, containing the numbers in each of its lines
DREPRESENTATION = *scalars*	Format to use for dates and times (stored in numerical structures)
MISSING = *texts*	Strings used to represent missing values in each text; default '*'
STATUS = *variates*	Code to indicate whether the number in each unit was read successfully (1), or with conversions (2), or unsuccessfully (0)

The TX2VARIATE directive forms variates from text structures. The texts are specified by the TEXT parameter, and are assumed to contain a single number in each of their strings. The variates are specified by the VARIATE parameter.

The DREPRESENTATION parameter specifies the format that has been used for texts that contain dates. For details, see 2.1.5. With the formats that specify only months and days, TX2VARIATE gives the number of the day within the year. However, it needs to know whether or not the year is a leap year. You can use the YEAR option to supply the year. If this is not set, TX2VARIATE assumes that it is not a leap year.

The MISSING parameter specifies a text for each text and variate, containing the string or strings that should be treated as missing values in the conversion; by default this is the string containing a single asterisk. Blank and null lines are always treated as missing.

By default, any non-numeric strings generate a missing value in the variate. However, you can set option NONNUMERIC=bestmatch to ignore commas, and to allow for the common typing errors that the letters i or l may have been typed instead of i, or that the letters o or O may have been typed instead of 0. You can set option PRINT=conversions to print a list of the values that have been converted. Also, the STATUS parameter can save a variate with a code for each number to show whether it was read successfully with no conversions (1), or only with conversions (2), or whether it could not be read successfully (0).

If you set option REDEFINE=yes, any data structure specified by the VARIATE parameter that is not a variate will be redefined (to be a variate). Also, VARIATE then takes the setting of the TEXT parameter as its default, i.e. it will redefine that text to be a variate.

This is illustrated in Example 4.5.3. Lines 2-6 show the various conversions, and lines 7-10 show how to use the DREPRESENTATION parameter to read dates.

Example 4.5.3

```
   2  TEXT       [VALUES=' 0.01',' -1','2.2','3.3E1',il,'-IO',O,'1,001',\
   3             ' ','*','notnumber!','3.3D2','1.23E-4','-1.23E'] Textvals
   4  TX2VARIATE [PRINT=conversions; NONNUMERIC=bestmatch] Textvals;\
   5             VARIATE=Realvals; Status=Status

Conversions
-----------

     String       Number
         il         11.0
        -IO        -10.0
          O          0.0
      1,001       1001.0
 notnumber!            *
     -1.23E         -1.2

   6  PRINT      Textvals,Realvals,Status; DECIMALS=*,6,0

  Textvals    Realvals      Status
      0.01    0.010000           1
        -1   -1.000000           1
       2.2    2.200000           1
     3.3E1   33.000000           1
        il   11.000000           2
       -IO  -10.000000           2
         O    0.000000           2
     1,001 1001.000000           2
                     *           1
         *           *           1
notnumber!           *           0
     3.3D2  330.000000           1
   1.23E-4    0.000123           1
    -1.23E   -1.230000           2

   7  TEXT       [VALUES='1/12/01','27/1/02','1/1/03','28/7/04','16/11/08',\
   8              '4/7/14','11/5/15','21/10/16','12/3/17','3/4/17'] Tdate
   9  TX2VARIATE Tdate; VARIATE=Vdate; DREPRESENTATION=3
  10  PRINT      Tdate,Vdate,Vdate; DREPRESENTATION=0,0,3

   Tdate       Vdate       Vdate
 1/12/01      146738     1/12/01
 27/1/02      146795     27/1/02
  1/1/03      147134      1/1/03
 28/7/04      147708     28/7/04
16/11/08      149280    16/11/08
  4/7/14      151336      4/7/14
 11/5/15      151647     11/5/15
21/10/16      152176    21/10/16
 12/3/17      152318     12/3/17
  3/4/17      152340      3/4/17
```

4.6 Operations on factors

You use factors in Genstat to indicate groupings of the units of vectors. You would need to do this, for example, in the analysis of designed experiments (Part 2 Chapter 4), or when forming tabular summaries of group totals, means, maxima, minima, and so on (4.11.1).

This section describes the GROUPS directive, which enables you to construct a factor from a variate or a text. The groups can cover every distinct value of the variate or text, or ranges of values; you can specify these ranges yourself, or have them defined automatically. Genstat can define the levels and labels vectors from either the minimum, the maximum or the median of the units allocated to each group.

Other facilities, for forming factors in experimental designs, are described elsewhere (2:4.9 and 2:4.13). The GENERATE directive (2:4.13.1) allows you to define factor values in a systematic order. You can also use it to form values of treatment factors, using the design-key method, or to define values for the pseudo-factors required to specify partially balanced experimental designs. Other facilities for generating factors in experimental designs are provided by the procedures in the Design module of the procedure library (2:4.9). The RANDOMIZE directive (2:4.11.1) can put the units of factors and variates into random order; this randomization can take account of the block structure of a designed experiment, if required.

The use of factors within expressions, and in CALCULATE in particular, is described in 4.1.1 and 4.1.2. This allows you to form a variate from a factor, either taking its declared levels or by taking an alternative set of levels using the NEWLEVELS function (4.2.1). CALCULATE also allows you to assign values of a variate to a factor, provided you have already declared the factor with levels including all the values taken by the variate. But a more satisfactory method is to use the GROUPS directive, already mentioned.

Procedure for manipulating factors include:

FACAMEND	permutes the levels and labels of a factor
FACCOMBINATIONS	forms a factor to indicate observations with identical values of a set of variates, texts or factors
FACDIVIDE	represents a factor by factorial combinations of a set of factors
FACEXCLUDEUNUSED	redefines the levels and labels of a factor to exclude those that are unused
FACLEVSTANDARDIZE	redefines a list of factors so that they have the same levels or labels
FACPRODUCT	forms a factor with a level for every combination of other factors
FACSORT	sorts the levels of a factor according to an index vector
FACUNIQUE	redefines a factor so that its levels and labels are unique
QFACTOR	allows the user to decide whether to convert texts or variates to factors

4.6.1 Forming factors from variates and texts: the GROUPS directive

GROUPS directive

Forms a factor (or grouping variable) from a variate or text, together with the set of distinct values that occur.

Options

PRINT = *string token*	Printed output required (summary); default * i.e. no printing
NGROUPS = *scalar*	Number of groups to form when LIMITS is not specified; if NGROUPS is also unspecified, each distinct value (allowing for rounding) defines a group; default *
LMETHOD = *string token*	Defines how to form the levels variate if the setting of the VECTOR parameter is a variate, or the labels if it is a text; if LMETHOD=* no levels/labels are formed, and existing levels (for a variate VECTOR) or labels (for a

	text VECTOR) of an already declared FACTOR will be retained if still appropriate (given, minimum, median, maximum, limit); default medi
DECIMALS = *scalar*	Number of decimal places to which to round the VECTOR before forming the groups; default * i.e. no rounding
BOUNDARIES = *string token*	Whether to interpret the LIMITS as upper or lower boundaries (upper, lower); default lowe
REDEFINE = *string token*	Whether to allow a structure in the FACTOR list that has already been declared (e.g. as a variate or text) to be redefined (yes, no); default no
CASE = *string token*	Whether the case of letters (small and capital) in text should be regarded as significant or ignored (significant, ignored); default sign
LDIRECTION = *string token*	How to define the levels (for a variate VECTOR) or labels (for a text VECTOR) when LMETHOD = minimum, median or maximum (ascending, given); default asce
OMITUNBOUNDED = *string token*	Whether to omit the (unbounded) group that occurs below the lowest limit when BOUNDARIES=lower, or above the final limit when BOUNDARIES=upper (yes, no); default no

Parameters

VECTOR = *variates* or *texts*	Vectors whose values are to define the groups
FACTOR = *factors*	Structures to be defined as factors to save details of the groups; default * will, if REDEFINE=yes, cause the corresponding VECTOR itself to be defined as a factor
LIMITS = *variates* or *texts*	Limits to define the groups
LEVELS = *variates*	Variate to define the levels of each FACTOR if LMETHOD=give, or to save them otherwise
LABELS = *texts*	Text to define the labels of each FACTOR if LMETHOD=give, or to save them otherwise

The GROUPS directive is designed to form factors from variates or texts. The variates and texts are specified by the VECTOR parameter, and the factors by the FACTOR parameter. With the simplest use of GROUPS you need specify no more than that, and each factor is defined to have a level for every distinct value of its corresponding variate or text. You need not have declared the factor already; it will be declared automatically if necessary.

Alternatively, you can divide the values of the variate or text into groups to be represented by the factor. You can use the LIMITS parameter to specify the range of values for each group. The limits vector is a text or a variate, depending whether the factor is being defined from a variate or a text; its values specify boundaries for the ranges. The BOUNDARIES option controls whether these are regarded as upper or lower boundaries; by default BOUNDARIES=lower. In Example 4.6.1 below, to divide the ages into the ranges 0-19, 20-29, 30-39, 40-49, 50-59 and over 60, the limits vector contains the five boundaries 20, 30, 40, 50 and 60. You can also ask GROUPS itself to set limits that will partition the units into groups of nearly equal size. You should then specify the NGROUPS option and leave the LIMITS parameter unset. (If you give both LIMITS and NGROUPS, then NGROUPS is ignored.)

If you are defining a factor from a variate VECTOR, the LMETHOD option controls how the levels vector is formed, with the following settings:

`median`	forms the levels from the median of the units in each group (default);
`minimum`	forms them from the minimum value in each group;
`maximum`	form them from the maximum value;
`limit`	uses the values in the `LIMITS` variate;
`given`	uses the values supplied (in a variate) by the `LEVELS` parameter.

With any of the settings `median`, `minumum`, `maximum` or `limit`, you can use the `LEVELS` parameter to specify a variate to store the levels that are produced; this can be done even if no factor is being formed, that is if no identifier is supplied for the factor by the `FACTOR` list. Finally, if you set `LMETHOD=*`, no levels are formed and any existing levels of the factor will be retained if they are still appropriate; otherwise the levels will be the integers 1 upwards. With any of these settings, you can use the `LABELS` parameter to specify labels for the factor.

Similar rules apply if you have a text `VECTOR` except that `LMETHOD` then governs how the labels are defined for the factor, and `LEVELS` can be used to specify its levels. The `CASE` option controls whether the case of the letters in the text strings is important. So, for example, if you set `CASE=ignored` the strings `'April'` and `'april'` will be put into the same group. With the default, `CASE=significant`, they would form different groups.

When the levels are formed from a `LIMITS` variate, there will be one group with no corresponding limit. If `BOUNDARIES=upper`, the extra group is above the final limit. The level assigned to that group is then the value that is the same distance above the final limit as the distance between the final limit and the last but one limit. If `BOUNDARIES=lower`, the extra group is below the first limit, and its level is given the value that is the same distance below the first limit as the distance between the first and second limits. The situation is similar with a `LIMITS` text, but the label for the extra group is always the single-character string `'-'`. If you would prefer to have an exact correspondence between the level and the limits, you can set option `OMITUNBOUNDED=yes` to omit the "unbounded" extra group. Any units beyond the final upper limit, or below the initial lower limit, are then given missing values.

The `LDIRECTION` option controls the ordering of the levels (for a variate `VECTOR`) or the labels (for a text `VECTOR`) when `LMETHOD` is set to `median`, `minimum` or `maximum`. By default, they are sorted into ascending order, but you can set `LDIRECTION=given` to take them in the order in which they occur in the `VECTOR`. This may be useful, for example, if a text vector contains the names of days or of months in calendar order.

You can set the `DECIMALS` option to request that the values of a variate `VECTOR` be rounded to a particular number of decimal places before the groups are formed: for example `DECIMALS=0` would round each value to the nearest integer.

You can redefine a `VECTOR` structure as a factor by setting option `REDEFINE=yes` and omitting to specify any corresponding identifier in the `FACTOR` list. This can be very useful on occasions when you are unable to define in advance which levels will occur in a set of data. In line 14 of Example 4.6.1, the text `National` (which contains details of the nationality of a list of people) is redefined as a factor so that we can produce a table with the mean ages of the people with each of the nationalities represented in the data.

Example 4.6.1

```
   2  VARIATE Age
   3  TEXT National
   4  READ National,Age

    Identifier   Minimum       Mean   Maximum     Values   Missing
      National                                        16         0
           Age     5.000      32.94     63.00         16         0

  11  GROUPS Age; FACTOR=Ageclass; LIMITS=!(20,30,40,50,60); \
     LABELS=!t('under 20','20-9','30-9','40-9','50-9','over 60')
   PRINT Age,Ageclass,National

         Age    Ageclass    National
       32.00        30-9     British
       29.00        20-9     British
        7.00    under 20     British
        5.00    under 20     British
       51.00        50-9      French
       49.00        40-9      French
       22.00        20-9     British
       24.00        20-9     British
       35.00        30-9     British
       41.00        40-9     British
       25.00        20-9      French
       24.00        20-9      French
       33.00        30-9     Italian
       29.00        20-9     Italian
       63.00     over 60     British
       58.00        50-9     British

   GROUPS [REDEFINE=yes] National
   TABULATE [CLASSIFICATION=National] Age; MEAN=MeanAge
   PRINT MeanAge

                  MeanAge
     National
      British        31.60
       French        37.25
      Italian        31.00
```

GROUPS takes account of any restrictions (4.4.1) on variates or texts in the VECTOR list, and will give missing values to the excluded units. If more than one vector is restricted, then each such restriction must be the same.

4.7 Operations on text

A text structure (2.3.2) is a vector each line of which contains a string of characters. So you might use it to label the units of other vectors, or to contain a complete piece of description.

The first part of this section describes the CONCATENATE directive (4.7.1) which allows you to concatenate several texts together side by side so that each line of the new text is formed by joining together a series of lines, one from each of the original texts. You can omit characters at the beginning and end of the component lines; so this also gives you a way of truncating the lines of a text. You can also change letters from upper to lower case, and vice versa.

Another form of concatenation (often known as *appending*) places whole texts one after another. You can do this with the APPEND and STACK procedures (4.4.4 and 4.4.5) or with the EQUATE directive (4.3.1) which also allows you to omit some of the lines.

The TXCONSTRUCT directive (4.7.2) is a more powerful (but more complicated) alternative to CONCATENATE, which allows you to concatenate textual representations of the values of variates, factors, scalars and pointers as well as lines of texts. You can again change case, and omit characters at the beginning and end of the component lines. You can also reverse the contents of the lines. Finally, you can choose to append the values from the texts, variates, factors, scalars or pointers, rather than concatenating them. It thus provides a very general set of facilities for text manipulation.

The TXPOSITION directive (4.7.3) allows you to search to see where a string of characters occurs within each of the lines of a text. Alternatively, you can use the TXFIND directive (4.7.4) to look for a subtext within a text, ignoring the line breaks (i.e. regarding them as space characters).

The TXREPLACE directive (4.7.5) replaces strings or subtexts within a text. To analyse a text in more detail, you can use the TXBREAK directive (4.7.6) to break it up into individual words.

Alternatively, the TXSPLIT procedure (4.7.7) splits a text into individual texts, at positions on each line marked by separator character(s). The TXPROGRESSION procedure allows you to form a text from a progression of character strings (4.7.9).

The remaining parts of the section describe the EDIT directive (4.7.10). This is a sub-system within Genstat; it has its own command syntax, allowing you to delete and insert series of characters, or to substitute one series for another, or to delete and insert complete lines, and so on.

Some general directives, described elsewhere, are also useful for manipulating text. The SORT directive allows you to sort the units of a text into alphabetical order or to form a factor from a text (4.4.3). You can test for equality and inequality of the lines of texts in the expressions that occur in CALCULATE (4.1), in RESTRICT (4.4.1) and in the directives for program control (5.2). CALCULATE also allows you to determine the number of characters in each line, and to find the positions of strings within lines (4.2.7). SETCALCULATE (4.3.3) can do Boolean arithmetic on the contents of text structures, and SETRELATE can compare their distinct sets of values. READ can take its input from a text (3.1.9), and you can direct output from the PRINT directive (3.2) into a text. PRINT thus allows you to place numerical values into a text. An alternative, for variates, is to use TXCONSTRUCT, which will also determine an appropriate number of decimal places.

4.7.1 Text concatenation: the CONCATENATE directive

CONCATENATE directive

Concatenates and truncates lines (units) of text structures; allows the case of letters to be changed.

Options

NEWTEXT = *text*	Text to hold the concatenated/truncated lines; default is the first OLDTEXT vector
CASE = *string token*	Case to use for letters (given, lower, upper, changed); default give leaves the case of each letter as given in the original string

Parameters

OLDTEXT = *texts*	Texts to be concatenated
WIDTH = *scalars* or *variates*	Number of characters to take from the lines of each text, a negative value takes all the (unskipped) characters other than trailing spaces; if * or omitted, all the (unskipped) characters are taken
SKIP = *scalars* or *variates*	Number of characters to skip at the left-hand side of the lines of each text, a negative value skips all initial spaces; if * or omitted, no characters are skipped

The CONCATENATE directive joins lines of several texts together, side by side, to form a new text. You can specify the identifier of this text by the NEWTEXT option, in which case it need not already have been declared as a text. If you do not specify NEWTEXT, Genstat places the new textual values into the first text in the OLDTEXT parameter list (replacing its existing values). The texts to be concatenated are specified by OLDTEXT. They should all contain the same number of lines, unless you want to insert an identical series of characters into every line of the new text. A series of characters that is to be duplicated within every line can be specified either as a string, or in a single-valued text. In line 8 of Example 4.7.1a, the string ', ' inserts a comma and a space into every line of the NEWTEXT Fullname.

Example 4.7.1a

```
   2  TEXT [VALUES='1. Adams','2. Baker','3. Clarke','4. Day', \
   3    '5. Edwards','6. Field','7. Good','8. Hall',\
   4    '9. Irving','10. Jones'] Name
   5  TEXT [VALUES='B.J.','J.S.','K.R.','A.T.','R.S.', \
   6    'T.W.','S.I.','D.M.','H.M.','C.C.'] Initials
   7  " Form text Fullname containing the number, name and initials."
   8  CONCATENATE [NEWTEXT=Fullname] OLDTEXT=Name,', ',Initials
   9  PRINT Fullname; JUSTIFICATION=left

Fullname
1. Adams, B.J.
2. Baker, J.S.
3. Clarke, K.R.
4. Day, A.T.
5. Edwards, R.S.
6. Field, T.W.
7. Good, S.I.
8. Hall, D.M.
9. Irving, H.M.
10. Jones, C.C.

  10  " Now reform Fullname to contain just the first initial and the name."
  11  CONCATENATE [NEWTEXT=Fullname] OLDTEXT=Initials,Name; \
  12    WIDTH=2,*; SKIP=*,!(9(2),3)
  13  PRINT Fullname; JUSTIFICATION=left

Fullname
B. Adams
J. Baker
K. Clarke
A. Day
R. Edwards
T. Field
S. Good
D. Hall
H. Irving
C. Jones
```

If you specify a variate in the `SKIP` list, it must contain a value for each line of the text in the `OLDTEXT` list; the value indicates the number of characters to be omitted at the beginning of that line. Alternatively, you can give a scalar if the same number of characters is to be omitted at the start of every line. In line 12 of the example, the null entry for `Initials` (indicated by `*`) specifies that no characters are to be omitted.

Similarly the `WIDTH` parameter specifies how many characters are to be taken, after omitting any initial characters as specified by `SKIP`. In line 12, `WIDTH` has a scalar setting of 2 for `Initials`, so that only the first initial followed by a dot is taken for each name. The `WIDTH` and `SKIP` parameters provide easy ways of removing spaces at the beginning or the end of strings. A negative value from the `SKIP` parameter deletes all the spaces at the start of a string, while a negative value from the `WIDTH` parameter deletes all the spaces at the end of a string. Example 4.7.1b illustrates the various possibilities: initial spaces are removed in forming the new texts `Trspace` and `Tnspace` (lines 7 and 8), and trailing spaces are removed from `Tlspace` and `Tnspace` (lines 6 and 8).

Example 4.7.1b

```
   2  TEXT [VALUES='1234567','    abc','abc    ','1234567'] Ts
   3  &    [VALUES=4('l-')] Tl
   4  &    [VALUES=4('-r')] Tr
   5  CONCATENATE [NEWTEXT=Tlrspace] Tl,Ts,Tr
   6  CONCATENATE [NEWTEXT=Tlspace]  Tl,Ts,Tr; WIDTH=2,-1,2
   7  CONCATENATE [NEWTEXT=Trspace]  Tl,Ts,Tr; SKIP=0,-1,0
   8  CONCATENATE [NEWTEXT=Tnspace]  Tl,Ts,Tr; WIDTH=2,-1,2; SKIP=0,-1,0
   9  PRINT Tlrspace,Tlspace,Trspace,Tnspace

   Tlrspace      Tlspace      Trspace      Tnspace
l-1234567-r l-1234567-r l-1234567-r l-1234567-r
l-    abc-r l-    abc-r     l-abc-r     l-abc-r
l-abc    -r     l-abc-r l-abc    -r     l-abc-r
l-1234567-r l-1234567-r l-1234567-r l-1234567-r
```

The `CASE` option enables you to change the case of letters. By default, `CASE=given` to leave the case of each letter as given in the existing text. To change all letters to upper case (or capitals) you can put `CASE=upper`, or `CASE=lower` to change all letters to lower case. Alternatively, `CASE=changed` puts lower-case letters into upper case, and upper-case letters into lower case!

`CONCATENATE` takes account of restrictions (4.4.1) on any of the vectors that occur in the statement. If more than one vector is restricted, then each such restriction must be the same. The values of the units that are excluded by the restriction are left unchanged.

4.7.2 Appending or concatenating values of scalars, variates, texts, factors or pointers: the `TXCONSTRUCT` directive

`TXCONSTRUCT` directive

Forms a text structure by appending or concatenating values of scalars, variates, texts, factors, pointers or formulae; allows the case of letters to be changed or values to be truncated and reversed.

Options

`TEXT` = *text*	Stores the text that is formed
`CASE` = *string token*	Case to use for letters (`given`, `lower`, `upper`, `changed`, `sentence`, `title`); default `give` leaves the case of each letter as given in the original texts
`METHOD` = *string token*	Whether to append or concatenate the values of the structures (`append`, `concatenate`) default `conc`
`SEPARATOR` = *string*	Characters to separate all except last two strings in each line when concatenating; default `''` (i.e. none)
`LASTSEPARATOR` = *string*	Characters to separate last two strings in each line when concatenating; default uses the characters defined by `SEPARATOR`
`PREFIX` = *string*	Characters to put at the start of each line when concatenating; default `''` (i.e. none)
`END` = *string*	Characters to put at the end of each line when concatenating; default `''` (i.e. none)
`SIGNIFICANTFIGURES` = *scalar*	Specifies the number of significant figures to include for numerical data; default 4

Parameters

`STRUCTURE` = *scalars*, *variates*, *factors*, *texts*, *pointers* or *formulae*	Structures whose values are to be appended or concatenated
`WIDTH` = *scalars* or *variates*	Number of characters to take from the strings formed from the units of each `STRUCTURE`, a negative value takes all the (unskipped) characters other than trailing spaces; if omitted or set to a missing value, all the

	(unskipped) characters are taken
`DECIMALS` = *scalars* or *variates*	Number of decimal places to use for numerical structures; if omitted or set to a missing value, a default is used which aims to print the value to the precision defined by the `SIGNIFICANTFIGURES` option
`SKIP` = *scalars* or *variates*	Number of characters to skip at the left-hand side of the strings formed from the units of each `STRUCTURE`, a negative value skips all initial spaces; if omitted or set to a missing value, no characters are skipped
`FREPRESENTATION` = *string tokens*	How to represent factor values (`labels, levels, ordinals`); default is to use `labels` if available, otherwise `levels`
`DREPRESENTATION` = *scalars* or *texts*	Format to use for dates and times (stored in numerical structures)
`REVERSE` = *string tokens*	Whether to reverse the strings of characters formed from the units of each structure (`yes`, `no`); default no
`MISSING` = *texts*	String to use to represent missing values of numerical structures; default `'*'`

The `TXCONSTRUCT` directive forms a text from the values of scalars, variates, texts, factors or pointers. The new text is saved using the `TEXT` option, and the structures from which it is to be formed are listed using the `STRUCTURE` parameter.

By default the values of the structures are concatenated alongside each other (as with the `CONCATENATE` directive); alternatively you can set option `METHOD=append` to append them below each other. When you are concatenating, the structures in the `STRUCTURE` list must generally contain the same number of values (and this then defines the number of lines in the new text). The exception is that the `STRUCTURE` list can include scalars or texts containing a single string if you want to put the same numbers or strings into every line of the new text.

Numerical values (from scalars, variates or factors) are converted into strings of characters before they are used. As in the `PRINT` directive, you can use the `DREPRESENTATION` parameter to indicate whether these are to be treated as dates. Alternatively, if they are to remain as numbers, the `DECIMALS` parameter specifies the number of decimal places to use. `DECIMALS` can be set to a scalar if all the values of the structure are to be printed with the same number of decimals, or to a variate if you want to represent different units of a variate or factor structure with different numbers of decimals. The `SIGNIFICANTFIGURES` option specifies the number of significant figures to aim for if `DECIMALS` is not set, or if it contains missing values (default 4). A numerical value will then be converted as though it had been printed with the number of decimals required to give `SIGNIFICANTFIGURES` significant figures, and any trailing zero decimal values had then been removed. Missing numerical values are represented by the asterisk character (`*`) by default, in the usual way, but you can specify another string of characters using the `MISSING` parameter.

A formula is converted to a text before being concatenated. The maximum width of the text is defined as 200. So this will have one line, unless the result is more than 200 characters wide.

The `SKIP` parameter allows you to skip characters at the start of the strings provided by each structure. You can supply a scalar to skip the same number of characters in every string, or a variate if you want to make different skips in every string. Similarly the `WIDTH` parameter specifies how many characters are to be taken, after omitting any initial characters as specified by `SKIP`. The strings formed from scalars, variates, factors and pointers do not contain any initial or trailing spaces. You can set a negative skip to ignore all the initial spaces in a string taken from a text structure, and set a negative width to ignore all its trailing spaces. The `REVERSE` parameter allows you to reverse the strings from any of the structures.

The `CASE` option enables you to change the case of letters in the strings. The available settings are:

`given`	to leave the case of each letter exactly as given in the string;
`upper`	to change all letters to upper case (or capitals);
`lower`	to change all letters to lower case;
`changed`	to put lower-case letters into upper case, and upper-case letters into lower case;
`sentence`	to put the first character in the text (if a letter) into upper case, then to use upper case only at the start of each new sentence;
`title`	to begin each new word with a capital letter, but otherwise to use lower case.

When `METHOD=concatenate` you can use the `SEPARATOR`, `LASTSEPARATOR`, `PREFIX` and `END` options to insert characters automatically between the adjacent pairs of strings in each line. `LASTSEPARATOR` supplies a string of characters to insert between the last pair of strings, `SEPARATOR` supplies characters to insert between all the other pairs of strings, `PREFIX` supplies characters to put at the start of each line, and `END` supplies characters to put at the end of each line. The defaults for `SEPARATOR`, `PREFIX` and `END` are the empty string `''`, while `LASTSEPARATOR` uses the characters defined by `SEPARATOR` as its default. So by default no characters are inserted.

`TXCONSTRUCT` takes account of restrictions on any of the vectors that occur in the statement. If more than one vector is restricted, then each such restriction must be the same. The values of the units in the new text that are excluded by the restriction are left unchanged.

Example 4.7.2a shows how `TXCONSTRUCT` extends `CONCATENATE`, by concatenating the ages to the names formed as in Example 4.7.1a. The `SEPARATOR` option can be left with its default value of a null text, as there will be a space at the start of each surname once the numbers (`'1.'`, `'2.'` and so on) are removed by the `SKIP` parameter. The LASTSEPARATOR option inserts the string `', age '` before the values from the variate `Age`. Notice that `TXCONSTRUCT` automatically determines that the Age values need no decimals.

Example 4.7.2a

```
VARIATE [VALUES=35,42,19,26,33,52,23,28,44,17] Age
TXCONSTRUCT [TEXT=Fullname; LASTSEPARATOR=', age '; END='.']\
  Initials,Name,Age; WIDTH=2,*,*; SKIP=*,!(9(2),3),*
PRINT Fullname; JUSTIFICATION=left

Fullname
B. Adams, age 35.
J. Baker, age 42.
K. Clarke, age 19.
A. Day, age 26.
R. Edwards, age 33.
T. Field, age 52.
S. Good, age 23.
D. Hall, age 28.
H. Irving, age 44.
C. Jones, age 17.
```

The `SEPARATOR` and `LASTSEPARATOR` options make it easy to construct a phrase from a text, as shown in Example 4.7.2b. Notice the use of the # symbol in lines 3 and 5 to insert the strings in Fruit as a list. In line 2, these are all separated by the string `' or '`, whereas in line 5 the last pair of strings is separated by `' and '` and the earlier pairs are separated by `', '`.

Example 4.7.2b

```
   2  TEXT [VALUES=apples,bananas,oranges,pears] Fruit
   3  TXCONSTRUCT [TEXT=List; SEPARATOR=' or '] #Fruit
   4  PRINT List

                              List
apples or bananas or orange or spears

   5  TXCONSTRUCT [TEXT=List; SEPARATOR=', '; LASTSEPARATOR=' and '] #Fruit
   6  PRINT List

                               List
apples, bananas, oranges and pears
```

4.7.3 Finding strings within the lines of a text structure: the `TXPOSITION` directive

`TXPOSITION` directive

Locates strings within the lines of a text structure.

Options

`CASE` = *string token*	Whether to treat the case of letters as significant when searching for lines of the `SUBTEXT` within the `TEXT` (`significant`, `ignored`); default `sign`
`REVERSE` = *string tokens*	Whether to reverse the search to work from the end of the lines of the `TEXT` (`yes`, `no`); default `no`
`MULTISPACES` = *string token*	Whether to treat differences between multiple spaces and single spaces as significant, or to treat them all like a single space (`significant`, `ignored`); default `sign`
`DISTINCT` = *string tokens*	Whether to require the `SUBTEXT` to have one or more separators to its left or right within the `TEXT` (`left`, `right`); default `*`
`SEPARATOR` = *text*	Characters to use as separators; default `' ,;:.'`

Parameters

`TEXT` = *texts*	Texts whose strings are to be searched
`SUBTEXT` = *texts*	Specifies a string or strings to find in each `TEXT`
`POSITION` = *variates*	Position of the `SUBTEXT` strings within the `TEXT`
`WIDTH` = *scalars* or *variates*	Right-most character(s) to search in the lines of each `TEXT`; default `*` searches up to the end of each line
`SKIP` = *scalars* or *variates*	Number of characters to skip at the left-hand side of the lines of each `TEXT`; default 0

The `TXPOSITION` directive allows you to search for strings of characters within the lines of a Genstat text structure. The text to search is specified by the `TEXT` parameter, and the `SUBTEXT` parameter specifies the strings that are to be found. You can set `SUBTEXT` to a single string (or to a text with just one line), if you want to search for the same string of characters within every line of the `TEXT`. You can set `SUBTEXT` to a text with as many lines as `TEXT`, if you want to search for different characters in each line of the `TEXT`. Finally, you can set `TEXT` to a single string, and `SUBTEXT` to a text with several lines, if you want to search the same string to see which of several strings might occur there. The `POSITION` parameter can save a variate storing the position of the first character of the `SUBTEXT` string(s) in each of the `TEXT` lines, or zero if the string has not been found.

TXPOSITION respects restrictions on any of the TEXT or SUBTEXT texts, and will search only the lines that are not excluded by the restriction. The values of the POSITION variate in the restricted units are left unchanged.

The SKIP parameter allows you to skip characters at the start of the lines of TEXT. You can supply a scalar to skip the same number of characters in every line, or a variate if you want to make different skips in each line. (So, once you have found a SUBTEXT string, you can set SKIP to its position and check whether it occurs again.) Similarly the WIDTH parameter specifies the right-most character(s) of the TEXT lines to search.

TXPOSITION usually takes account of the case of letters (small or capital) when looking for the SUBTEXT strings within the TEXT. So for example 'GenStat' would not match with 'Genstat'. However, you can set option CASE=ignored to ignore differences in case. It will usually also treat multiple spaces as significant, but you can set option MULTISPACE=ignored to treat them all like a single space.

Option DISTINCT is useful if you are looking for distinct words or phrases. The left setting requires each SUBTEXT string to begin either at the start of the relevant line of TEXT, or to be preceded in that line by a separator (such as a space or comma). Similarly, the right setting requires the SUBTEXT to end within the line of TEXT with a separator (or to be at the end of the line). The separators are specified by the SEPARATOR option.

Example 4.7.3 first uses TXPOSITION to find the lines of the text Intro6 that contain the string 'Genstat'. These are indicated by the non-zero units of the variate Where. Then it sets the SKIP option of TXPOSITION to skip these first occurrences of 'Genstat', so that it can find the lines where 'Genstat' occurs a second time. These are indicated by non-zero units of the variate Next.

Example 4.7.3

```
TEXT Intro6; VALUES=!t(\
'Genstat has very comprehensive facilities for Analysis of Variance.',\
'Almost all of these can be accessed using custom menus. In this',\
'chapter, we start with the simplest design, a one-way completely',\
'randomized experiment, before introducing factorial experiments,',\
'which have more than one treatment or fixed effect. We use an',\
'experiment with a randomized block design to show how to deal with',\
'blocks, which involve more than one stratum or source of error in',\
'the analysis, and extend this idea by analysing a split-plot design.',\
'Many other types of design can also be analysed by Genstat, and',\
'details are available in Chapter 4 of Part 2 of the Guide to',\
'Genstat. We also introduce some of Genstat''s extensive facilities',\
'for creating designed experiments, available from the Design option',\
'of the Stats menu.')
TXPOSITION Intro6; SUBTEXT='Genstat'; POSITION=Where
TXPOSITION Intro6; SUBTEXT='Genstat'; POSITION=Next; SKIP=Where
PRINT      Where,Next; DECIMALS=0

   Where        Next
       1           0
       0           0
       0           0
       0           0
       0           0
       0           0
       0           0
       0           0
      52           0
       0           0
       1          36
       0           0
       0           0
```

4.7.4 Finding a subtext within a text: the TXFIND directive

TXFIND directive

Finds a subtext within a text structure.

Options

CASE = *string token*	Whether to treat the case of letters (small or capital) as significant when searching for the SUBTEXT within the TEXT (significant, ignored); default sign
REVERSE = *string token*	Whether to reverse the search to work from the end of the TEXT (yes, no); default no
MULTISPACES = *string token*	Whether to treat differences between multiple spaces and single spaces as significant, or to treat them all like a single space (significant, ignored); default sign
DISTINCT = *string tokens*	Whether to require the SUBTEXT to have one or more separators to its left or right within the TEXT (left, right); default *
SEPARATOR = *string*	Characters to use as separators; default ' ,;:.'
SAMELINE = *string token*	Whether to ignore matches in the TEXT where the SUBTEXT is not all on the same line (yes, no); default no

Parameters

TEXT = *texts*	Texts to be searched
SUBTEXT = *texts*	Text to look for in each TEXT
COLUMN = *scalars*	Position of the column within TEXT where the first character of SUBTEXT has been found
LINE = *scalars*	Number of the line within TEXT where the first character of SUBTEXT has been found
ICOLUMN = *scalars*	Column within TEXT at which to start the search
ILINE = *scalars*	Line within TEXT at which to start the search
ENDCOLUMN = *scalars*	Position of the column within TEXT where the last character of SUBTEXT has been found
ENDLINE = *scalars*	Number of the line within TEXT where the last character of SUBTEXT has been found

The TXFIND directive looks for a Genstat text structure within another text structure. The text to search is specified by the TEXT parameter, and the SUBTEXT parameter specifies the text to be found. By default, the search treats the OLDTEXT and OLDSUBTEXT as if they were paragraphs of characters: that is, it takes no account of the line breaks within the two text structures, regarding each one as equivalent to a space. However, you can set option SAMELINE=yes to ensure that matches will be recognised only if they are all on a single line. Any restrictions on the texts are ignored.

The COLUMN parameter saves the column within the TEXT where the first character of the SUBTEXT is found, and the LINE parameter saves its line within the TEXT. These are both set to zero if SUBTEXT is not found. Similarly the ENDCOLUMN and ENDLINE parameters save the position of the last character of the SUBTEXT. You can use the ICOLUMN and ILINE parameters to specify a starting column and line for the search. So you can search for the next occurrence of SUBTEXT by setting ILINE to the saved value of LINE, and ICOLUMN to the saved value of COLUMN plus one.

TXFIND usually takes account of the case of letters (small or capital) when looking for the

SUBTEXT within the TEXT. So for example 'Genstat' would not match with 'Genstat'. However, you can set option CASE=ignored to ignore differences in case. It will usually also treat multiple spaces as significant, but you can set option MULTISPACE=ignored to treat them all like a single space.

Option DISTINCT is useful if you are looking for distinct words or phrases. The left setting requires the SUBTEXT to begin either at the start of the TEXT, or to be preceded in the TEXT by a separator (such as a space or comma). Similarly, the right setting requires the SUBTEXT to end within the TEXT with a separator (or to be at the end of the TEXT). The separators are specified by the SEPARATOR option.

Example 4.7.4 searches the text Intro6 from Example 4.7.3 to find all instances of the word 'the'. The option setting DISTINCT=left,right ensures that 'the' is a distinct word (so it does not find 'the' within the word 'these' in line 2). The FOR and ENDFOR directives, which look over the commands to find the second and subsequent instances of 'the', are described in 5.2.1. The EXIT directive, which exits the loop if 'the' is not found, is described in 5.2.4.

Example 4.7.4

```
      TXFIND     [DISTINCT=left,right] Intro6; SUBTEXT='the';\
                 COLUMN=column; LINE=line
      PRINT      [SQUASH=yes] line,column & Intro6$[line] & '!'; FIELD=column
        line      column
       3.000       24.00
chapter, we start with the simplest design, a one-way completely
                       !
      FOR [NTIMES=999]
        TXFIND   [DISTINCT=left,right] Intro6; SUBTEXT='the';\
                 COLUMN=column; LINE=line; ICOLUMN=column+1; ILINE=line
        EXIT     line .EQ. 0
        PRINT    [SQUASH=yes] line,column & Intro6$[line] & '!'; FIELD=column
      ENDFOR
        line      column
       8.000       1.000
the analysis, and extend this idea by analysing a split-plot design.
!
        line      column
       10.00       49.00
details are available in Chapter 4 of Part 2 of the Guide to
                                                !
        line      column
       12.00       51.00
for creating designed experiments, available from the Design option
                                                  !
        line      column
       13.00       4.000
of the Stats menu.
   !
```

4.7.5 Replacing a subtext within a text: the TXREPLACE directive

TXREPLACE directive

Replaces a subtext within a text structure.

Options

NTIMES = *scalar*	Number of times to search for the OLDSUBTEXT and replace it; default 1
CASE = *string token*	Whether to treat the case of letters (small or capital) as significant when searching for the OLDSUBTEXT within the OLDTEXT (significant, ignored); default sign
MULTISPACES = *string token*	Whether to treat differences between multiple spaces

	and single spaces as significant when locating the OLDSUBTEXT within the OLDTEXT, or to treat them all like a single space (significant, ignored); default sign
DISTINCT = *string tokens*	Whether to require the OLDSUBTEXT to have one or more separators to its left or right within the OLDTEXT (left, right); default *
SEPARATOR = *string*	Characters to use as separators; default ' ,;:.'
SAMELINE = *string token*	Whether to ignore matches in the OLDTEXT where the OLDSUBTEXT is not all on the same line (yes, no); default no

Parameters

OLDTEXT = *texts*	Texts to be edited
NEWTEXT = *texts*	Texts with OLDSUBTEXT replaced by NEWSUBTEXT; if no NEWTEXT is supplied, the new values replace those in the corresponding OLDTEXT
OLDSUBTEXT = *texts*	Text to look for in each OLDTEXT
NEWSUBTEXT = *texts*	Text to replace OLDSUBTEXT
COLUMN = *scalars*	Position of the column within OLDTEXT where the first character of NEWSUBTEXT has been placed
LINE = *scalars*	Number of the line within OLDTEXT where the first character of NEWSUBTEXT has been placed
ICOLUMN = *scalars*	Column within OLDTEXT at which to start the search
ILINE = *scalars*	Line within OLDTEXT at which to start the search
ENDCOLUMN = *scalars*	Position of the column within OLDTEXT where the last character of NEWSUBTEXT has been placed
ENDLINE = *scalars*	Number of the line within OLDTEXT where the last character of NEWSUBTEXT has been placed
NREPLACED = *scalars*	Number of subtexts replaced

The TXREPLACE directive replaces a subtext within a Genstat text structure. The text containing the subtext is specified by the OLDTEXT parameter. The OLDSUBTEXT parameter specifies the subtext to be replaced, and the NEWSUBTEXT parameter specifies the subtext to replace it. By default, as in TXFIND (4.7.4), the search treats the OLDTEXT and OLDSUBTEXT as if they were paragraphs of characters: that is, it takes no account of the line breaks within the two text structures, regarding each one as equivalent to a space. However, you can set option SAMELINE=yes to treat line breaks differently from spaces. Matches are then recognised only if they are all on a single line. Any restrictions on the texts are ignored.

By default a single occurrence of the subtext is replaced, but you can use the NTIMES option to replace several. It you set NTIMES to a negative value, all occurrences are replaced. The NREPLACED parameter can save the number of replacements that were actually made (which may be less than NTIMES if fewer were found in the OLDTEXT). The new text (after the replacements) can be saved using the NEWTEXT parameter; if this is not set, the values of the OLDTEXT are replaced by the new text.

TXREPLACE usually takes account of the case of letters (small or capital) when looking for the OLDSUBTEXT within the OLDTEXT. So for example 'Genstat' would not match with 'Genstat'. However, you can set option CASE=ignored to ignore differences in case. It will usually also treat multiple spaces as significant, but you can set option MULTISPACE=ignored to treat them all like a single space.

Option DISTINCT is useful if you are looking for distinct words or phrases. The left setting requires the OLDSUBTEXT to begin either at the start of the OLDTEXT, or to be preceded in the OLDTEXT by a separator (such as a space or comma). Similarly, the right setting requires the OLDSUBTEXT to end within the OLDTEXT with a separator (or to be at the end of the OLDTEXT). The separators are specified by the SEPARATOR option.

The ICOLUMN and ILINE parameters can specify a starting column and line for the search. So you can leave an initial section of the OLDTEXT unchanged.

You can use the COLUMN parameter to save the column within the OLDTEXT where the first character of the NEWSUBTEXT has been inserted, and the LINE parameter to save its line within the OLDTEXT. These are both set to zero if the OLDSUBTEXT was not found. If NTIMES is greater than one, they save the location of the final replacement. Similarly the ENDCOLUMN and ENDLINE parameters can save the position of the last character of the NEWSUBTEXT within the OLDTEXT.

TXREPLACE is illustrated in Example 4.7.5. Notice that, as option SAMELINE is left with its default setting of no, the string 'baked beans' is replaced by 'salad', even though it is split over two lines.

Example 4.7.5

```
TEXT Lunch; VALUES=!t(\
'For lunch we will have fish fried in batter, with chips and baked',\
'beans, followed by apple pie with ice cream.')
TXREPLACE Lunch; NEWTEXT=Healthierlunch;\
          OLDSUBTEXT=' fried in batter'; NEWSUBTEXT=''
&         Healthierlunch; NEWTEXT=Healthierlunch;\
          OLDSUBTEXT='chips'; NEWSUBTEXT='new potatoes'
&         Healthierlunch; NEWTEXT=Healthierlunch;\
          OLDSUBTEXT='baked beans'; NEWSUBTEXT='salad'
&         Healthierlunch; NEWTEXT=Healthierlunch;\
          OLDSUBTEXT=' with ice cream'; NEWSUBTEXT=''
PRINT     Lunch; JUSTIFICATION=left

Lunch
For lunch we will have fish fried in batter, with chips and baked
beans, followed by apple pie with ice cream.

&         Healthierlunch; JUSTIFICATION=left

Healthierlunch
For lunch we will have fish, with new potatoes and salad, followed by apple pie.
```

4.7.6 Extracting the individual words from a text: the TXBREAK directive

TXBREAK directive

Breaks up a text structure into individual words.

Option

SEPARATOR = *text*	Defines the characters separating the words in the original text; default ' ,;:.'

Parameters

TEXT = *texts*	Text to break into words
WORDS = *texts*	Saves the words contained in each text (in the order in which they occur)
COLUMNS = *variates*	Saves the number of the column in the TEXT where each word began
LINES = *variates*	Saves the number of the line where each word was found

`PLACESINLINES` = *variates*	Saves the place of each word (first, second &c) within the line where it was found

The `TXBREAK` directive forms a text containing all the words (including duplicates) found in a text. The original text to break up is supplied by the `TEXT` parameter, and the `WORDS` parameter saves a text storing the words that it contains. The words are stored in the order in which they occur in the original text (but, for example, you could use the `SORT` directive to sort them into alphabetic order). The `LINES` parameter can save a variate recording the line in the original text where each one was found. The `COLUMNS` parameter can save a variate recording the column where each word began, and the `PLACESINLINES` parameter can save a variate giving the place of each word (first, second &c) within the line where it was found.

By default, the words are assumed to be separated from one another by spaces or by any of the standard punctuation characters (comma, semi-colon, colon, full stop). However, you can use the `SEPARATOR` option to specify some other characters. For example, you could put `SEPARATOR=' ,;:.?'` to allow question marks as well. These characters are all removed from the words when they are stored.

`TXBREAK` takes account of any restrictions on the original text, and omits the words in the restricted lines.

Example 4.7.6 uses `TXBREAK` to form the text `Words` containing all the words in the text `Intro6` from Examples 4.7.3 and 4.7.4. It then uses the `GROUPS` directive (4.6.1) to convert `Words` to a factor, and the `TABULATE` directive (4.11.1) to count how many times each word occurs.

Example 4.7.6

```
  28  TXBREAK  Intro6; WORDS=Words
  29  GROUP    [CASE=ignored; REDEFINE=yes] Words
  30  TABULATE [PRINT=count; classification=Words]

                     Count
        Words
            2            1
            4            1
            a            3
     accessed            1
          all            1
       Almost            1
         also            2
           an            1
     analysed            1
    analysing            1
     Analysis            2
          and            2
          are            1
    available            2
           be            2
       before            1
        block            1
       blocks            1
           by            2
          can            2
      chapter            2
   completely            1
comprehensive            1
     creating            1
       custom            1
         deal            1
       design            5
     designed            1
      details            1
       effect            1
        error            1
   experiment            2
```

```
 experiments          2
      extend          1
   extensive          1
  facilities          2
   factorial          1
       fixed          1
         for          2
        from          1
     Genstat          3
   Genstat's          1
       Guide          1
         has          1
        have          1
         how          1
        idea          1
          In          3
   introduce          1
 introducing          1
     involve          1
        Many          1
        menu          1
       menus          1
        more          2
          of          8
         one          2
     one-way          1
      option          1
          or          2
       other          1
        Part          1
  randomized          2
        show          1
    simplest          1
        some          1
      source          1
  split-plot          1
       start          1
       Stats          1
     stratum          1
        than          2
         the          5
       these          1
        this          2
          to          3
   treatment          1
       types          1
         use          1
       using          1
    Variance          1
        very          1
          we          3
       which          2
        with          3
```

4.7.7 Splitting a text vertically into individual texts: the TXSPLIT procedure

TXSPLIT procedure

Splits a text into individual texts, at positions on each line marked by separator character(s) (R.W. Payne).

Options

SEPARATOR = *text*	Defines the character(s) that indicate where to split each line; default `','`
INCLUDE = *string tokens*	Whether to retain the separator at the end of a split text, or any spaces at its start and end (`separators`, `spaces`) ; default `*` i.e. include neither

Parameters

`TEXT` = *texts*	Text to split
`SPLITTEXTS` = *texts*	Saves the texts into which `TEXT` is split

`TXSPLIT` splits a text into individual texts. The positions at which to split each line are marked by the character, or characters, specified by the `SEPARATOR` option; by default, the separator character is a comma.

By default, `TXSPLIT` removes the separators between the split texts, as well as any spaces at the start and end of each spit text (i.e. any spaces around the separators, or at the start or end of the original text). The `INCLUDE` option allows you to request that the separator be left at the end of a split text, and that these spaces should be retained.

The `TEXT` parameter supplies the text that is to be split. The texts into which it is split are saved, in a pointer, by the `SPLITTEXTS` parameter. Any restrictions on the original text are ignored.

Example 4.7.7 continues Example 4.7.1a. Each line of the the text `Fullname` contains the name, followed by a comma and a space, and then the age. Line 20 splits `Fullname` into a text containing the name, and another containing the age, using the default separator (comma and space).

Example 4.7.7

```
  18  " Split Fullname into a text containing the name and another
 -19    containing the age (using the default separator ', ')."
  20  TXSPLIT Fullname; SPLITTEXTS=Split
  21  PRINT Split[]; JUSTIFICATION=left

Split[1]   Split[2]
B. Adams   age 35.
J. Baker   age 42.
K. Clarke  age 19.
A. Day     age 26.
R. Edwards age 33.
T. Field   age 52.
S. Good    age 23.
D. Hall    age 28.
H. Irving  age 44.
C. Jones   age 17.
```

4.7.8 Integer codes for textual characters: the `TXINTEGERCODES` directive

`TXINTEGERCODES` directive

Converts textual characters to and from their corresponding integer codes.

Options

`CONVERTTO` = *string token*	Whether to convert from text characters to integer codes or integer codes to text characters (`codes`, `text`) ; default `code`
`REPRESENT` = *string token*	How to treat code values 128-255 (`extendedascii`, `utf8`); default `exte` if `CODES` defines no characters that can be represented only in UTF-8, otherwise `utf8`

Parameters

`TEXT` = *texts*	Text structures (each with a single line only)

`CODES` = *variates* or *scalars*	Integer codes corresponding to the characters in each text

Textual characters all have corresponding integer code values (see http://unicode.org/charts/). For example, the characters in the basic ASCII character set have codes running from 0 to 127. The letters a-z have codes 97-122, the capital letters have codes 65-90, and the digits 0-9 have codes 48-57. These characters can all be represented by a single "byte" of computer storage, consisting of eight "bits" each able to store either one or zero. Genstat stores other characters, such as those in the Chinese, Korean or Thai languages, in the UTF-8 format which uses up to four bytes per character.

By default, `TXINTEGERCODES` takes as input a text supplied by the `TEXT` parameter, which must contain only one line. The codes corresponding to the characters in the line are saved in a variate, supplied by the `CODES` parameter. Alternatively, if you set option `CONVERTTO` = `text`, the codes are taken as input, and `TEXT` saves the corresponding line of characters. Missing or zero codes are ignored, and invalid codes (for example, negative numbers) are faulted.

Codes 128-255 can be represented either by characters in the extended ASCII character set, or by 2-byte UTF-8 characters. These represent the same actual characters, but you may find one representation more convenient than the other, depending on how you want to use any output involving the text in future. If you have a preference, you can control this by setting the `REPRESENT` option. Otherwise, `TXINTEGERCODES` uses extended ASCII characters, unless the variate contains codes that can be represented only in UTF-8.

Example 4.7.8 shows the integer codes for some European landmarks. The UTF-8 format is used for the text `Finished` in line 9, as the output in lines 2 and 3 contains UTF-8 characters for the Greek and Russian names.

Example 4.7.8

```
  2  TEXT     [VALUES='Ο Παρθενώνας'] Parthenon
  3  &        [VALUES='Красный квадрат'] RedSquare
  4  &        [VALUES='Château de Versailles'] Versailles
  5  TXINTEGERCODES Parthenon,RedSquare,Versailles; CODES=Pcodes,Rcodes,Vcodes
  6  PRINT    Pcodes,Rcodes,Vcodes; DECIMALS=0

     Pcodes       Rcodes       Vcodes
        927         1050           67
         32         1088          104
        928         1072          226
        945         1089          116
        961         1085          101
        952         1099           97
        949         1081          117
        957           32           32
        974         1082          100
        957         1074          101
        945         1072           32
        962         1076           86
                    1088          101
                    1072          114
                    1090          115
                                   97
                                  105
                                  108
                                  108
                                  101
                                  115

  7  VARIATE [VALUES=84,111,117,116,32,101,115,116,32,\
  8          116,101,114,109,105,110,233] Fcodes
  9  TXINTEGERCODES [CONVERTTO=text; REPRESENT=utf8] Finished; CODES=Fcodes
 10  PRINT    Finished
```

```
       Finished
Tout est terminé
```

4.7.9 Progressions of character strings: the **TXPROGRESSION** procedure

TXPROGRESSION procedure

Forms a text containing a progression of strings (R.W. Payne).

Options

`INCLUDECHARACTERS` = *string tokens*	Defines the set of characters to include in the progression (`lower`, `upper`, `digits`, `_`, `%`, `space`); default `lowe`
`DIRECTION` = *string token*	Direction of the progression (`ascending`, `descending`); default `asce`
`FIRSTLETTERS` = *string token*	Controls which letters come first (`alllower`, `allupper`, `lower`, `upper`); default `uppe`
`OWNCHARACTERSET` = *text*	Can supply an alternative set of characters

Parameters

`FIRST` = *texts*	Single-valued text specifying the first string in each progression
`SECOND` = *texts*	Single-valued text specifying the second string in each progression
`LAST` = *texts*	Single-valued text defining the end of each progression
`PROGRESSION` = *texts*	Saves the progression

`TXPROGRESSION` forms a text from a progression of strings. This is saved by the `PROGRESSION` parameter. It could be used, for example, for labels of factors (2.2.3), or for defining rows and columns of matrices (2.4.1).

The `INCLUDECHARACTERS` option specifies the characters to include in the progression, with settings:

`lower`	for lower-case letters (a-z);
`upper`	for upper-case letters (A-Z);
`digits`	for the numerical characters 0-9;
`_`	for the underscore character;
`%`	for the percent character;
`space`	for the space character.

If they are all specified, the characters will appear in the order: space, percent, digits 0-9, underscore, and then letters. The default is to include only lower-case letters. The alternative, if you do not like any of these possibilities, is to specify your own set of characters, using the `OWNCHARACTERS` option.

The `FIRSTLETTERS` option controls the ordering of lower- and upper-case letters, if both are included, with settings:

`alllower`	all lower-case letters first;
`allupper`	all upper-case letters first;
`lower`	letters interspersed, in pairs, with the lower-case letter first (i.e. a, A, b, B etc.);
`upper`	letters interspersed, in pairs, with the upper-case letter first (i.e. A, a, B, b etc.).

The default is `upper`.

The `DIRECTION` option specifies whether the progression is in ascending order (e.g. a-z) or descending order (e.g. z-a). Ascending order is the default.

The first string in the progression is specified by the `FIRST` parameter. The `SECOND` parameter can supply the second string in the progression, thus defining the increment between the strings. If this is not specified, the default is to increment the right-hand character in the string by one for an ascending progression, and minus one for a descending progression. The `LAST` parameter defines the end of the progression. (The progression stops when the next string would go beyond `LAST`.) `FIRST`, `SECOND` and `LAST` must all contain the same number of characters.

Example 4.7.9 forms a progression in which three letters are each incremented by one, starting at `abc`, so that they shift through the alphabet one letter at a time.

Example 4.7.9

```
   2  TXPROGRESSION 'abc'; SECOND='bcd'; LAST='xyz'; PROGRESSION=Shiftedletters
   3  PRINT Shiftedletters

Shiftedletters
           abc
           bcd
           cde
           def
           efg
           fgh
           ghi
           hij
           ijk
           jkl
           klm
           lmn
           mno
           nop
           opq
           pqr
           qrs
           rst
           stu
           tuv
           uvw
           vwx
           wxy
           xyz
```

4.7.10 Editing text: the `EDIT` directive

The `EDIT` directive provides a line editor for modifying text structures.

`EDIT` directive

Edits text vectors.

Options

`CHANNEL` = *scalar* or *text*	Text structure containing editor commands or a scalar giving the number of a channel from which they are to be read; default is the current input channel
`END` = *text*	Character(s) to indicate the end of the commands read from an input channel; default is the character colon (`:`)
`WIDTH` = *scalar*	Limit on the line width of the text; default `*`
`SAVE` = *text*	Text to save the editor commands for future use; default `*`

Parameters

`OLDTEXT` = *texts*	Texts to be edited
`NEWTEXT` = *texts*	Text to store each edited text; if any of these is omitted, the corresponding `OLDTEXT` is used

The `EDIT` directive edits each text in the `OLDTEXT` list, storing the results in the corresponding structure in the `NEWTEXT` list. It both edits and stores each text before moving on to the next. If you have not already declared any of the texts in the `NEWTEXT` list, it will be declared implicitly. If you give a missing identifier (`*`) in the `NEWTEXT` list, the edited version simply replaces the values of the original: thus the old text will be overwritten by the new text. You can also omit a text from the `OLDTEXT` list; you might do this if you wanted to form the values of the new text entirely from within the editor. If any of the old texts are restricted, they must all be restricted to exactly the same set of units. Then only those units will be involved in the edit. When a restriction is in force, you cannot add or delete any units (or lines).

The `CHANNEL` option tells Genstat where to find the editing commands. A scalar specifies the number of an input channel from which the commands are to be read. Alternatively, you can specify a text structure containing the commands. In either case the commands should be terminated by the string specified by the `END` option. The end string can be more that one character; the default is the single character colon (`:`). Genstat gives a warning if you have forgotten to specify the end string in a text of commands. The default for the `CHANNEL` option is to take input from the current input channel.

The `WIDTH` option specifies the maximum line length for vectors of commands and of text, the default being 80 and the maximum being 255.

The `SAVE` option allows you to specify a text structure to store the edit commands, so that you can save them for future `EDIT` statements.

You can give commands to the editor in upper or lower case. You can put as many commands as you like on a line, subject only to the width restriction set by the `WIDTH` option. Commands must be separated by at least one space. You cannot put spaces into the middle of a command, unless they are part of a character string (or part of a sequence of commands).

The character that separates the parts of a command is written here as `/`, but you can use any character for this other than a space or a digit.

Genstat puts the lines from the old text into an internal *buffer*, where they are modified according to the commands that you specify. While you are editing, Genstat moves a notional *marker* around the buffer. The marker can be moved backwards or forwards along a line or between lines. So you can move around the text and modify the lines in any order. Some commands move the marker automatically, as explained in the definitions below. If the marker is before the first line of text it is at the `[start]` position; if it is after the last line of text it is at the `[end]` position. The line that currently contains the marker is called the *current line*. Genstat does not write anything to the new text until the edit has been completed (so if you use the `Q` command, the new text is left unaltered).

Some commands allow you to specify a number: for example `D`*n* deletes the next *n* lines. Genstat gives a warning message if this number is zero or is not an integer.

The command definitions are as follows.

`A`	Insert the next line of text from the buffer, immediately after the marker within the current line.
`B`	Break the current line at the marker position. Text before the marker is written as a new line to the internal buffer and text after the marker becomes the new current line with the marker at character position 1.
`C`	Cancel edits performed on the current line by restoring it to the form in which it was most recently read from the buffer. Note that if you have previously edited the line and then moved to some other line, it is the previously edited form that will be given, not the form

as originally in the old text; also, if you have given any `A` or `B` commands during your modification of the current line, their effects are not negated, so for example any lines that have been inserted into the current line by `A` will be lost.

`D`	Delete the current line, and make the next line the current line with the marker at character position 1.
`D`*n*	Delete the next *n* lines (including the current line), making the next line after that the current line with the marker at position 1.
`D+`*n*	Synonymous with `D`*n*.
`D+`	is a synonym for `D` or `D+1`.
`D+*`	Delete from the current line to end of text. The current line is then `[end]`.
`D*`	Synonymous with `D+*`.
`D-`	Delete the current line, making the previous line the current line with the marker at character position 1.
`D-`*n*	Delete the current and previous *n* lines, making the line before that the current line with the marker at character position 1.
`D-`	is a synonym for `D-1`.
`D-*`	Delete the current line and all previous lines, the current line is then `[start]`.
`D/`*s*`/`	Delete from the current line to the line with the next occurrence of the character string *s*. The marker is placed immediately before the character string *s* in the located line. If *s* occurs after the marker on the current line, the marker is moved up to *s* and no lines are deleted.
`D-/`*s*`/`	The same as `D/`*s*`/`, except that it moves backwards through the text, deleting all lines from and including the current one until the first occurrence of a line containing the character string *s*. The marker is placed immediately before the located character string *s*. If *s* occurs before the marker on the current line, the marker placed before *s* and no lines are deleted.
`F/`*i*`/`	Inserts the contents of the text structure with identifier *i* immediately before the current line. The marker is not moved.
`G+/`*s*`/`*t*`/`	substitutes string *t* for all occurrences of string *s* found after the marker on the current and subsequent lines, and moves the marker to the end of the text.
`G/`*s*`/`*t*`/`	is a synonym for `G+/`*s*`/`*t*`/`.
`G-/`*s*`/`*t*`/`	substitutes string *t* for all occurrences of string *s* found before the marker on the current and previous lines, and moves the marker to the start of the text.
`I/`*s*`/`	Inserts the string *s* as a new line immediately before the current line. The marker is not moved.
`L`	Moves the marker to the start of the next line, which can be `[end]`.
`L`*n*	Moves the marker to the start of the *n*th line after the current line. So `L1` gives the next line.
`L+`*n*	Is synonymous with `L`*n*.
`L+`	Is synonymous with `L` or `L+1`.
`L+*`	Moves the marker to `[end]`.
`L*`	is a synonym for `L+*`.
`L-`*n*	Moves the marker to the start of the *n*th line before the current line, which can be `[start]`. `L-1` gives the line immediately before the current line.
`L-`	Is synonymous with `L-1`.
`L-*`	Moves the marker to `[start]`.
`L+/`*s*`/`	Moves the marker to the position immediately before the next occurrence of the character string *s* after the current marker position; this occurrence need not be on the current line. If the string *s* is not found, the marker will be located at `[end]`.
`L-/`*s*`/`	Moves the marker to the position immediately before the first occurrence of the string *s* before the current marker position; this occurrence need not be on the current line. If the

string *s* is not found, the marker will be located at `[start]`.

`P` moves the marker one character to the right along the current line.

`P+`*n* Moves the marker *n* characters to the right of the current position within the current line. You cannot move the marker beyond the maximum line length (which will vary between computers, but is normally the same as the width of your local line-printer).

`P+` is a synonym for `P` or `P+1`.

`P+*` Moves the marker to the position immediately after the last non-blank character in the current line. This can be to the left of the current marker position.

`P-`*n* Moves the marker *n* characters to the left of the current position within the current line. The marker cannot be moved to the left of character position 1.

`P-` is a synonym for `P-1`.

`P-*` Moves the marker to the position immediately before the first non-blank character after character position 1. This can be to the right of the current marker position.

`P`*n* Moves the marker to the character position *n* within the current line, counting from the left and starting at 1. The maximum value of *n* varies between computers but is normally the same as the width of your local line-printer.

`Q` Abandons the current edit, leaving the original text unaltered.

`R+/`*s*`/`*t*`/` substitutes character string *t* for the next occurrence of character string *s* after the marker on the current or subsequent lines, and moves the marker to the position immediately after *t*.

`R/`*s*`/`*t*`/` is a synonym for `R+/`*s*`/`*t*`/`.

`R-/`*s*`/`*t*`/` substitutes string *t* for the nearest occurrence of string *s* before the marker on the current or previous lines; the marker moves to be immediately before string *t*.

`S/`*s*`/`*t*`/` Substitutes the string *t* for the next occurrence of string *s* after the marker within the current line. The marker is moved to the character position immediately after the last character in *t*. If *s* is null (when the command is `S//`*t*`/`) then *t* is inserted immediately after the marker. If *t* is null (when the command is `S/`*s*`//`), then *s* is deleted from the line.

`V` Turns on the verification mode. Then, if you are working interactively, the current line will be displayed each time that Genstat prompts you for commands. By default the marker is indicated by the character > but you can change this by the command `V`*c* or `V+`*c*.

`V`*c* Turns on the verification mode (see `V`), and changes the marker character to *c*.

`V+`*c* Is synonymous with `V`*c*.

`V-` Turns verification mode off (see `V`).

`(`*cseq*`)` *n* Repeats the command sequence, *cseq*, *n* times. The command sequence *cseq* can be any valid combination of editing commands, each separated by at least one space. The complete sequence, including brackets and repeat count, must all be on a single line. You can nest sequences up to a depth of 10.

`(`*cseq*`) *` Repeats the command sequence cseq until `[end]` or `[start]` is encountered. In all other respects `(`*cseq*`) *` behaves exactly as `(`*cseq*`)` *n*; so it would be equivalent to putting *n* equal to some very large number.

Example 4.7.10

```
> " An interactive run of the editor."
> TEXT Name
> OPEN 'Names.dat'; CHANNEL=2
> READ [CHANNEL=2] Name

    Identifier   Minimum      Mean   Maximum    Values   Missing

          Name                                      10         0

> " Edit Name: within the editor the prompt will be 'EDIT> '."
> EDIT Name
```

```
>B.J. Adams
EDIT> S//Mr. /
Mr. >B.J. Adams
EDIT> L
>J.S. Baker
EDIT> (S//Dr. / L)4
>T.W. Field
EDIT> S//Ms. /
Ms. >T.W. Field
EDIT> L
>S.I. Good
EDIT> S//Mr. /
Mr. >S.I. Good
EDIT> L S//Miss. /
Miss. >D.M. Hall
EDIT> (L S//Dr. /)2
Dr. >C.C. Jones
EDIT> :
> PRINT Name; JUSTIFICATION=left

Name
Mr. B.J. Adams
Dr. J.S. Baker
Dr. K.R. Clarke
Dr. A.T. Day
Dr. R.S. Edwards
Ms. T.W. Field
Mr. S.I. Good
Miss. D.M. Hall
Dr. H.M. Irving
Dr. C.C. Jones
```

4.8 Operations on formulae and expressions

If you are writing procedures, for example for statistical analyses, the model to be fitted will often be specified by a Genstat formula structure (2.2.4). Unless the algorithm within the procedure merely involves straightforward use of one of Genstat's statistical directives, you may wish to know more about the formula: how many model terms does it contain, which factors do they involve, and so on. The FCLASSIFICATION directive (4.8.1) is designed to provide the answers to these questions.

Genstat expression structures (2.2.3) are also used frequently in procedures, to specify calculations, and again you may want to find out what data structures are being used. The FARGUMENTS directive (4.8.2) allows you to obtain lists of the data structures that are involved in the calculation, and those that will store the results.

Another useful directive is SET2FORMULA (4.8.3), which provides a convenient way of constructing standard formulae involving a specified set of factors and variates. Alternatively, the REFORMULATE directive can take a "template" formula or expression, and modify it to refer to a particular collection of data structures (4.8.4).

4.8.1 The FCLASSIFICATION directive

FCLASSIFICATION directive

Forms a classification set for each term in a formula, breaks a formula up into separate formulae (one for each term), and applies a limit to the number of factors and variates in the terms of a formula.

Options

FACTORIAL = *scalar*	Limit on the number of factors and variates in each term; default * i.e. no limit

NTERMS = *scalar*	Outputs the number of terms in the formula
CLASSIFICATION = *pointer*	Saves a list of all the factors and variates in the TERMS formula
OUTFORMULA = *formula structure*	Identifier of a formula to store a new formula, omitting terms with too many factors and variates
INCLUDEFUNCTIONS = *string token*	Whether or not to include functions in the formulae saved by the OUTFORMULA option or the OUTTERMS parameter (yes, no); default no
REORDER = *string token*	When to reorder the terms in the model (always, standard, never); default stan
DROPTERMS = *string token*	Whether to include only terms that can be dropped individually from the formula (yes, no); default no
CHECKFUNCTIONS = *scalar*	Indicator, set to one if the TERMS formula contains any functions, and zero if it contains none
FUNCTIONDEFINITIONS = *pointer*	Saves details of the functions defined for each factor and variate in the TERMS formula

Parameters

TERMS = *formula*	Formula from which the classification sets, individual model terms and so on are to be formed
CLASSIFICATION = *pointers*	Identifiers for pointers to store the factors and variates composing each model term of the TERMS formula
OUTTERMS = *formula structures*	Identifiers for formulae to store each individual term of the TERMS formula
MAINTERMS = *formula structures*	Identifiers for formulae to store the main term for each individual term of the TERMS formula

The FCLASSIFICATION directive enables you to manipulate a formula data structure; the formula is specified using the TERMS parameter.

As explained in 1.6.3, when Genstat uses a formula in a statistical analysis, it is expanded into a series of model terms, linked by the operator +. FCLASSIFICATION allows you to save this expanded form, in another formula, using the OUTFORMULA option.

You can use the FACTORIAL option to apply a limit to the number of factors and variates in the resulting terms, similarly to the FACTORIAL option in the ANOVA, REML and regression directives (2:4.1.2, 2:5.3.1 and 2:3.3.1). The number of terms in the formula can be saved (in a scalar) using the NTERMS option, and a list of the factors and variates that occur in the formula can be saved (in a pointer) using the CLASSIFICATION option.

The other parameters allow you to save information about the individual model terms in the formula. The identifiers in the lists that they specify are taken in parallel with the model terms in the expanded form of the formula. For each model term, the corresponding identifier in the list for the CLASSIFICATION parameter is defined as a pointer storing the factors that occur in the term. The identifier in the OUTTERMS list is defined as a formula containing just that model term. The MAINTERMS parameter is useful if the formula contains pseudo-factors. Its identifiers save formula structures containing the "main term" for each of the model terms. If the term is a pseudo-term, this will be the model term to which the pseudo-term is linked. Otherwise, it will be the term itself. For example, in the model

```
    Variety//(A+B)
```

in Example 2:4.7.3c, there are two pseudo-terms, A and B, with Variety as their main term.

By default any functions such as POL or REG are omitted from the formulae saved by OUTFORMULA or OUTTERMS, but these will be included if you set option INCLUDEFUNCTIONS=yes. The CHECKFUNCTIONS option allows you to save a scalar containing one if the `TERMS` formula contains any functions, and zero if it does not.

The `FUNCTIONDEFINITIONS` option allows you to obtain details of the functions. This saves a pointer which contains a pointer for each factor and variate in the formula (in the same order as in the `CLASSIFICATION` pointer). If the factor or variate has no function, its pointer contains just a text with a single missing value (`''`). Otherwise the first element of the pointer is a text containing the name of the function (either `'POL'`, `'POLND'`, `'REG'`, `'REGND'`, `'COMP'`, `'SSPLINE'` or `'LOESS'`). It then contains elements to store the second and subsequent arguments of the function (if any).

Model terms involving several factors are regarded by Genstat as representing all the joint effects of these factors that are not removed by earlier terms in the formula. So, in the formula

```
A + B + A.B
```

`A.B` is the interaction of factors A and B, as both main effects occur earlier in the formula. Alternatively, in the formula

```
A.B + A + B
```

`A.B` still represents all the joint effects of factors `A` and `B`, and the later terms `A` and `B` are redundant as they are now "contained" in `A.B`. Thus `FCLASSIFICATION` usually deletes any term in the model that is contained in an earlier term. However, if you set option `REORDER=always`, the model is reordered after applying any operator (including plus). The reordering arranges the terms so that they contain increasing numbers of identifiers. Terms with the same number of identifiers are then put into lexicographical order with respect to the order in which the identifiers first occurred in the formula itself. Each term will therefore come before any term that would contain it. So the model would again be

```
A + B + A.B
```

The default setting, `REORDER=standard`, applies the standard Genstat rules, which reorder the terms only after a dot, slash or star operator. The final setting `REORDER=never` specifies that no reordering should take place. (Before Release 19.2, the `ORTHOGONAL` option had settings `no` and `yes`, corresponding to `standard` and `always`. Options and parameters with settings `yes` and `no` should not have any other settings. So these were renamed in Release 19.2, when the setting `never` was added. However, `no` and `yes` are retained as synonyms, so that earlier programs will still run.)

The rules about terms that contain other terms are also relevant when you are dropping terms from a model, for example in a regression analysis. You cannot drop a term, for example using the `DROP` directive, until all the terms that contain it have been dropped. To simplify the process, if you set option `DROPTERMS=yes`, the formulae saved by `OUTFORMULA` or `OUTTERMS` will contain only terms that are not contained in any other terms (i.e. only the terms that can be dropped).

The use of `FCLASSIFICATION` is illustrated in Example 4.8.1. At line 3, formula `ABC2` is formed, to contain the expanded form of the formula `A*B*C` subject to the limit of `FACTORIAL=2`. Lines 5 and 7 obtain information about the individual terms in the formula. In line 5, the `NTERMS` option is used to ascertain how many terms there are. The resulting scalar, `NT`, can then be used in line 7 to specify the necessary lists of identifiers: `Class[1...NT]` for the `CLASSIFICATION` parameter, and `Term[1...NT]` for the `OUTTERMS` parameter.

Example 4.8.1

```
   2  FACTOR [NVALUES=32; LEVELS=2] A,B,C
   3  FCLASSIFICATION [FACTORIAL=2; OUTFORMULA=ABC2] A*B*C
   4  PRINT ABC2

        ABC2
A + B + C + A.B + A.C + B.C
   5  FCLASSIFICATION [FACTORIAL=2; NTERMS=NT] A*B*C
   6  PRINT NT
```

```
          NT
       6.000

FCLASSIFICATION [FACTORIAL=2] A*B*C; CLASSIFICATION=Class[1...NT]; \
  OUTTERMS=Term[1...NT]
FOR Ci=Class[]; Oi=Term[]
  PRINT [SERIAL=yes] Ci,Oi
ENDFOR


    Class[1]
           A


      Term[1]
A


    Class[2]
           B


      Term[2]
B


    Class[3]
           C


      Term[3]
C


    Class[4]
           A
           B


      Term[4]
A.B


    Class[5]
           A
           C


      Term[5]
A.C


    Class[6]
           B
           C


      Term[6]
B.C
```

4.8.2 The `FARGUMENTS` directive

`FARGUMENTS` directive

Forms lists of arguments involved in an expression.

Options

`EXPRESSION` = *expression structure*
Expression whose arguments are required

`NRESULTS` = *scalar*	Number of results generated by the expression
`NCALCULATIONS` = *scalar*	Number of calculations in the expression

Parameters

`ICALCULATION` = *scalars*	The calculation from which to save the result and arguments
`RESULT` = *dummies*	Stores the result structure for calculation `ICALCULATION`
`ARGUMENTS` = *pointers*	Stores the arguments in calculation `ICALCULATION`

If you are writing a procedure that takes an expression as one of its inputs, you may want to know what results it is generating and what data structures it is using to calculate them. The `FARGUMENTS` allows you to find this out.

The expression to study is specified by the `EXPRESSION` option. The `NRESULTS` option can save the number of results, and the `NCALCULATIONS` option can save the number of calculations. The parameters of `FARGUMENTS` allow you to save information about each of the calculations in the expression: the `ICALCULATION` parameter specifies the number of the calculation, the `RESULT` parameter can specify a dummy to be set to the structure that is given the result, and the `ARGUMENTS` parameter can specify a pointer to save the arguments.

The use of `FARGUMENTS` is illustrated in Example 4.8.2. The first part of the example examines, `Sum`, a fairly simple expression. The dummy `SumRes` is set to the data structure `S`, that result of the sum. Notice that we need to put `SumRes` into an unnamed pointer to print its contents (line 4). The pointer `SumArgs` contains the four data structures `A`, `B`, `C` and `D`, whose values are summed. The second part of the example examines, `Transformationm`, an expression that contains two calculations (each with a result). First we use the `NRESULTS` option to find the number of results (which will be the same as the number of calculations); see line 8. Then we obtain the result and arguments for each calculation. Notice that the arguments can be unnamed data structures, like the scalar constant 1. This does not have an identifier, so it generates a blank line when the pointer TArgs[i] is printed by line 13. However, you can see in the output from line 14, which prints the contents of the arguments.

Example 4.8.2

```
   2  EXPRESSION Sum; VALUE=!e(S=A+B+C+D)
   3  FARGUMENTS [EXPRESSION=Sum] 1; RESULT=SumRes; ARGUMENTS=SumArgs
   4  PRINT      !p(SumRes)

           S

   5  PRINT      SumArgs

     SumArgs
           A
           B
           C
           D

   6  SCALAR     L1,L2,P1,P2
   7  EXPRESSION Transformation; VALUE=!e(L1,L2=LOG(P1,P2/(1-P1,P2)))
   8  FARGUMENTS [EXPRESSION=Transformation; NRESULTS=Nt]
   9  FARGUMENTS [EXPRESSION=Transformation] 1...Nt; RESULT=TRes[1...Nt];\
  10             ARGUMENTS=TArgs[1...Nt]
  11  FOR [NTIMES=Nt; INDEX=i]
  12    PRINT    !p(TRes[i])
  13    PRINT    TArgs[i]
  14    PRINT    TArgs[i][]
  15  ENDFOR
```

```
         L1


   TArgs[1]
         P1

         P1


         P1                  P1
          *      1.000        *



         L2


   TArgs[2]
         P2

         P2


         P2                  P2
          *      1.000        *
```

4.8.3 The **SET2FORMULA** directive

SET2FORMULA directive

Forms a model formula using a set of structures supplied in a pointer.

Option

METHOD = *string token*	Relationship of the structures within the formula (combined, crossed, nested); default comb

Parameters

POINTER = *pointers*	Sets of structures to be used to form the formulae
FORMULA = *formula structures*	Formulae constructed from the sets

SET2FORMULA forms a model formula using the contents (factors and/or variates) of a pointer. The pointer is specified by the POINTER parameter, and the formula is saved by the FORMULA parameter.

The METHOD option defines how the formula is constructed. With the combined setting, the formula has a single model term combining all the structures: for example

```
SET2FORMULA !p(A,B,C); FORMULA=Fcomb
```

sets up Fcomb as the formula

```
A.B.C
```

The crossed setting links the contents of the pointer using the operator *, so it would form the formula

```
A*B*C
```

The nested setting uses the operator /, so it would form

```
A/B/C
```

4.8.4 The **REFORMULATE** directive

REFORMULATE directive

Modifies a formula or an expression to operate on a different set of data structures.

Options

OLDFORMULA = *formula* or *expression structure*	Original formula or expression
NEWFORMULA = *formula* or *expression structure*	New formula or expression, modified to operate on the new structures

Parameters

OLDSTRUCTURE = *identifiers*	Data structures in the OLDFORMULA to be replaced in the NEWFORMULA
NEWSTRUCTURE = *identifiers*	Identifier of the new data structure to replace each OLDSTRUCTURE

The REFORMULATE directive is useful if you have a "template" formula or expression which you would like to customize to operate on a particular collection of data structures. The template formula or expression is specified by the OLDFORMULA option, and the customized formula or expression is specified by the NEWFORMULA option. If NEWFORMULA is not specified, the customized formula or expression replaces the old one in OLDFORMULA. The data structures to be replaced in OLDFORMULA are listed by the OLDSTRUCTURE parameter, and the corresponding data structures for NEWFORMULA are provided by the NEWSTRUCTURE parameter.

The statements below show how you could convert formula

```
F1 + F2 * F3
```

(stored in Old) into formula

```
Blocks + A * B
```

(stored in New).

```
FORMULA      [VALUE=F1 + F2 * F3] Old
REFORMULATE  [OLDFORMULA=Old; NEWFORMULA=New]\
             OLDSTRUCTURE=F1,F2,F3; NEWSTRUCTURE=Blocks,A,B
```

4.9 Operations on dummies and pointers

You use dummies (2.2.2) when you want the same series of statements to operate on different data structures on different occasions. By referring to a dummy instead of any specific structure, you can make the statements apply to whichever structure you want. The commonest use of dummies is in loops (5.2.1) and in procedures (5.3).

In this section we describe an alternative way of specifying a value for a dummy, by using the ASSIGN directive (4.9.1). ASSIGN also enables you to change the values of elements of pointers, which are used mainly to specify collections of data structures for directives such as EQUATE (4.3.1), or as a convenient way of specifying lists of structures (1.5.4 and 2.6). A further use of ASSIGN is to control the labelling of structures that exist as subscripted identifiers of two or more pointers but which do not possess identifiers in their own right (see Example 4.9.1c).

You can make tests on the values of dummies and pointers using the .IS. and .ISNT. operators (4.1.1).

4.9.1 Assigning values to dummies and individual elements of pointers: the ASSIGN directive

ASSIGN directive

Sets elements of pointers and dummies.

Options

`NSUBSTITUTE` = *scalar*	Number of times *n* to substitute a dummy in order to determine which structure to assign (if *n* is negative, the assigned structure is the -*n*th from the bottom of the chain of dummies, like the `NTIMES` option of `EXIT`); default 0 i.e. no substitution
`METHOD` = *string token*	Whether to replace or preserve the existing value in each dummy or pointer element (`replace`, `preserve`); default `repl` (note, pointer elements are never unset so `METHOD=preserve` with a pointer simply causes the assignment to be ignored)
`RENAME` = *string token*	Whether to reset the default name for the structure if it has only a suffixed identifier (`yes`, `no`); default `no`
`SCOPE` = *string token*	This allows dummies or pointer elements within a procedure to be set to point to structures in the program that called the procedure (`SCOPE=external`) or in the main program itself (`SCOPE=global`) rather than to structures within the procedure (`local`, `external`, `global`); default `loca`
`NSTRUCTURESUBSTITUTE` = *scalar*	Number of times *n* to substitute a dummy setting of the `STRUCTURE` parameter in order to determine which structure to assign to the setting of the `POINTER` parameter (if *n* is negative, the assigned structure is the -*n*th from the bottom of the chain of dummies, like the `NTIMES` option of `EXIT`); default 0 i.e. no substitution

Parameters

`STRUCTURE` = *identifiers*	Values for the dummies or pointer elements
`POINTER` = *dummies* or *pointers*	Structure that is to point to each of those in the `STRUCTURE` list
`ELEMENT` = *scalars* or *texts*	Unit or unit label indicating which pointer element is to be set; if omitted, the first element is assumed

`ASSIGN` allows you to set individual elements of pointers, or to assign a value to a dummy. The parameter `POINTER` lists the pointers or dummies whose values you want to set; the values that you want to give them are listed by the `STRUCTURE` parameter. You pick out the individual elements of pointers by the `ELEMENT` parameter; a scalar identifies the element by its suffix number, while a text identifies it by its label. This example sets the dummy `Yvar` to point to the variate `Height`, and elements 1 and 2 of the pointer `Xvars` to `Protein` and `Vitamins`, respectively.

```
VARIATE Height,Protein,Vitamins
POINTER [NVALUES=2] Xvars
DUMMY   Yvar
ASSIGN  Height,Protein,Vitamins;POINTER=Yvar,2(Xvars); \
        ELEMENT=1,1,2
```

Element 1 is assumed unless you specify otherwise; so to set just `Yvar` we need only put

```
ASSIGN Height; POINTER=Yvar
```

Options `METHOD`, `NSUBSTITUTE` and `NSTRUCTURESUBSTITUTE` are likely to be most useful when setting dummies within a procedure. By setting `METHOD=preserve`, any dummies that are already set will have their existing settings preserved. Hence this provides a very convenient and effective way of making default assignments while leaving any explicit assignments unchanged. Suppose, for example, that a procedure has dummy arguments `FITTEDVALUES`, `RESIDUALS` and `RSS` available to save various aspects of the analysis, and that we wish to use these as working variables while calculating this information within the procedure. By specifying

```
ASSIGN [METHOD=preserve] LocalF,LocalR,LocalRSS; \
  FITTEDVALUES,RESIDUALS,RSS
```

any of the dummies that is not set when the procedure is called will be assigned to the corresponding local structure, either `LocalF`, `LocalR` or `LocalRSS`. Note, however, that elements of pointers cannot be unset; they will always point to some identifier, even if it is unnamed. Thus, `ASSIGN` has no effect on elements of pointers when `METHOD=preserve`.

The `NSUBSTITUTE` option is useful when you have dummies pointing to other dummies, in a chain. This may happen when one procedure calls another, passing one of its own arguments as the argument to the procedure that it calls. The `NSUBSTITUTE` option allows the dummies in the `POINTER` list to be substituted a set number of times in order to determine which dummy in a chain is to be assigned a value.

When the procedure `ARGDEF` is called from procedure `ADDONE` at line 16 of Example 4.9.1a, the dummy `ARG` which is the first parameter of procedure `ARGDEF`, points to the dummy `RESULT` which is the second parameter of procedure `ADDONE`. (See 5.3 for more details about procedures.) By setting option `NSUBSTITUTE=1` in line 6 of the example, the dummy `ARG` is substituted once before it is assigned, so that the value is assigned to the dummy `RESULT`. Notice that this option affects only the dummies in the `POINTER` list, and not any that appear elsewhere; thus the dummy `DEFAULT` will be substituted to the variate `Xplus1` to which `DEFAULT` is set at line 16.

Example 4.9.1a

```
  2  PROCEDURE 'ARGDEF'
  3  "Assigns a default to an unset dummy argument."
  4    PARAMETER NAME='ARG','DEFAULT'; MODE=p; TYPE='dummy',*
  5    IF UNSET(ARG)
  6      ASSIGN [NSUBSTITUTE=1] DEFAULT; ARG
  7    ENDIF
  8  ENDPROCEDURE
  9
 10  PROCEDURE 'ADDONE'
 11  "Adds one to the values of variate X and prints results
-12   (which can also be saved using the RESULTS parameter)."
 13    PARAMETER 'X','RESULT'; MODE=p; SET='yes','no'; DECLARED='yes','no';\
 14      TYPE='variate'; COMPATIBLE=!t(nvalues,type); PRESENT='yes','no'
 15    VARIATE   Xplus1
 16    ARGDEF    RESULT; DEFAULT=Xplus1
 17    CALCULATE RESULT=X+1
 18    PRINT     RESULT
 19  ENDPROCEDURE
 20
 21  VARIATE [VALUES=1,3,5,7] Y
 22  ADDONE  Y
```

```
Xplus1
 2.000
 4.000
 6.000
 8.000
```

Sometimes it may be easier to specify which dummy to assign by counting up from the bottom of the chain of dummies, instead of down from the top. You should then set NSUBSTITUTE to a negative integer. In Example 4.9.1b, dummy A points to dummy B, which in turn points to dummy C, and dummy C then points to dummy D, which points to the scalar X (line 3). Thus, at line 4

```
ASSIGN [NSUBSTITUTE=-1] Y; A
```

will assign Y to the dummy one from the bottom of the chain, that is C, and so

```
PRINT C,D
```

at line 5, prints the values of Y and X.

Example 4.9.1b

```
   2  SCALAR X,Y; VALUE=1,2
   3  DUMMY  A,B,C,D; VALUE=B,C,D,X
   4  ASSIGN [NSUBSTITUTE=-1] Y; A
   5  PRINT C,D

            Y           X
        2.000       1.000
```

Similarly, the NSTRUCTURESUBSTITUTE option is useful when you have a dummy as the setting of the STRUCTURE parameter. By default, it is the dummy itself that is assigned to the corresponding dummy or pointer in the POINTER list. However, you can set NSTRUCTURESUBSTITUTE, in the same way as NSUBSTITUTE, to substitute the dummy before making the assignment.

The RENAME option enables you to control what identifier is used for data structures in the rare occasions when your program contains structures that can be referred to by more than one suffixed identifier and which do not have identifiers in their own right. This is illustrated in Example 4.9.1c.

At line 3 of Example 4.9.1c, pointer P is defined to have two elements: suffix 1 refers to the scalar X and suffix 2 to the scalar Y. Line 4 introduces the identifier P[3], so Genstat expands P to have a third suffix; but this structure can be referred to only as P[3] – as is shown when the three structures are printed in line 5. Line 6 defines a new pointer, Q, and sets its values to be the same as those of P; P[3] can now be referred to as Q[3] but, when Genstat prints this structure, it uses the original identifier P[3] (see line 8). Line 9 shows another way of defining the values of Q, using ASSIGN. At the same time, we can change the identifier that Genstat will then use, by setting option RENAME=yes. This is confirmed when Q[3] is printed in line 10.

Example 4.9.1c

```
   2  SCALAR X,Y; VALUE=1,2
   3  POINTER [VALUES=X,Y] P
   4  SCALAR P[3]; VALUE=3
   5  PRINT P[]

            X           Y        P[3]
        1.000       2.000       3.000

   6  POINTER [VALUES=P[1,2,3]] Q
   7  CALCULATE Q[1,2,3] = Q[1,2,3]*10
```

```
  8  PRINT Q[]

          X           Y        P[3]
      10.00       20.00       30.00

  9  ASSIGN [RENAME=yes] P[3]; POINTER=Q; ELEMENT=3
 10  PRINT Q[]

          X           Y        Q[3]
      10.00       20.00       30.00
```

Finally, the SCOPE option enables you to assign a dummy within a procedure to a structure in the program that called the procedure. The dummy will thus operate as though it was a dummy option or parameter, except that the decision about the structure that it references in the outer program has been made within the procedure instead of outside it. This facility allows you to define new data structures in the outer program; however, care needs to be taken to ensure that there is no conflict with any existing structures.

4.10 Operations on matrices and compound structures

The CALCULATE directive (4.1.1 and 4.1.3) allows you to do arithmetic operations on matrices element by element: addition, subtraction, multiplication, division and exponentiation, as well as logical operations of testing for equality and inequality, and so on; you can also do matrix multiplication. There are several functions for standard operations on matrices such as taking inverses, for forming various standard types of matrix or for constructing matrices from tables (4.2.4). You can combine and omit rows or columns of a rectangular matrix using the COMBINE directive (4.11.4). EQUATE allows you to transfer values to matrices from another structure, and vice versa (4.3.1), or you can select sub-matrices with CALCULATE, using qualified identifiers (4.1.6). Procedures that operate on matrices include: LINDEPENDENCE to find the linear relations associated with matrix singularities; FHADAMARDMATRIX to form Hadamard matrices; FPROJECTIONMATRIX to form a projection matrix for a set of model terms; PARTIALCORRELATIONS to form a matrix of partial correlation coefficients from a set of variates; FCORRELATION forms and tests the correlation matrix for a list of variates; FVCOVARIANCE to form a variance-covariance matrix from a set of variates; and ROBSSPM to form robust estimates of sums-of-squares-and-products matrices. Ordinary correlations can be formed using the CORRELATE directive (which also forms autocorrelations of variates and lagged cross-correlations between variates).

You cannot do calculations directly with a complete compound structures like an LRV or an SSPM, but you can do calculations with the individual elements. For example, to take the diagonal matrix of latent roots from an LRV structure, L, and divide it by the trace, you could put

```
    CALCULATE L['Roots'] = L['Roots'] / L['Trace']
```

This section describes the SVD, FLRV and QRD directives, which allow you to form singular value, eigenvalue and QR decompositions; it also describes the FSSPM directive, which calculates sums of squares and products and all the associated information stored in an SSPM structure. These operations form the basis of many common statistical methods. Another directive that operates on compound structures is the FTSM directive, which forms preliminary values of a time-series model in a TSM structure.

4.10.1 The singular value decomposition: the `SVD` directive

`SVD` directive

Calculates singular value decompositions of matrices.

Option

`PRINT` = *string tokens*	Printed output required (`left, singular, right`); default `*` i.e. no printing

Parameters

`INMATRIX` = *matrices*	Matrices to be decomposed
`LEFT` = *matrices*	Left-hand matrix of each decomposition
`SINGULAR` = *diagonal matrices*	Singular values (middle) matrix
`RIGHT` = *matrices*	Right-hand matrix of each decomposition

Suppose that we have a rectangular matrix A with m rows and n columns, and that p is the minimum of m and n. The singular value decomposition can be defined as

$$ {}_mA_n \;=\; {}_mU_p \; {}_pS_p \; {}_pV_n' $$

The diagonal matrix S contains the p singular values of A, ordered such that

$$ s_1 \geq s_2 \geq \dots \geq s_p \geq 0 $$

The matrices U and V contain the left and right singular vectors of A, and are orthonormal:

$$ U'U = V'V = I_p $$

The smaller of U and V will be orthogonal. So, if A has more rows than columns, m>n, p=n and VV'=I_p.

The least-squares approximation of rank r to A can be formed as

$$ A_r = U_r S_r V_r' $$

where U_r and V_r are the first r columns of U and V, and S_r contains the first r singular values of A (Eckart & Young 1936).

The `INMATRIX` parameter specifies the matrices to be decomposed. The algorithm uses Householder transformations to reduce A to bi-diagonal form, followed by a QR algorithm to find the singular values of the bi-diagonal matrix (Golub & Reinsch 1971). The other parameters allow you to save the component parts of the decomposition: `LEFT`, `SINGULAR` and `RIGHT` for U, S and V respectively.

The `PRINT` option allows you to print any of the components of the decomposition; by default, nothing is printed. If any of the matrices is to be printed, all p columns are shown, even if you are storing only the first r columns. See Example 4.10.1a.

Example 4.10.1a

```
   2  MATRIX [ROWS=6; COLUMNS=4; VALUES=\
   3    15,5,9,16,3,20,7,12,22,17,10,11,13,8,1,23,2,4,6,14,18,21,24,19] A
   4  SVD [PRINT=LEFT,SINGULAR,RIGHT] A

Singular value decomposition
============================

Singular values
---------------

                          1           2           3           4
                      65.30       17.75       14.29       10.82
```

```
Left singular vectors
---------------------

                  1           2           3           4
          1     0.35066    -0.33717    -0.30338     0.26324
          2     0.32642     0.30654     0.69925    -0.39495
          3     0.45861     0.18847    -0.51086    -0.52922
          4     0.37075    -0.71706     0.15091    -0.29662
          5     0.20711    -0.27069     0.36641     0.39876
          6     0.61629     0.41157    -0.03151     0.49765

Right singular vectors
----------------------

                  1           2           3           4
          1     0.50011    -0.13783    -0.80932    -0.27549
          2     0.50254     0.53368     0.40553    -0.54607
          3     0.40479     0.48070    -0.09456     0.77210
          4     0.57749    -0.68199     0.41426     0.17258
```

Genstat will decide how many columns and singular values *r* to store, and will store that number for any of the components that you specify. If none of the matrices in the `LEFT`, `SINGULAR` and `RIGHT` lists has been declared in advance, the full number of singular values ($r=p$) is stored; otherwise Genstat sets *r* to the maximum number of columns contained in any of the matrices. If $r<p$, the first *r* singular values will be saved, along with the corresponding columns of singular vectors.

Example 4.10.1b

```
   5  DIAGONALMATRIX [ROWS=2] Sa
   6  SVD A; LEFT=Ua; SINGULAR=Sa; RIGHT=Va
   7  CALCULATE A2 = Ua *+ Sa *+ TRANSPOSE(Va)
   8  PRINT [RLWIDTH=6] A; FIELDWIDTH=9; DECIMALS=3

             A
             1        2        3        4

    1   15.000    5.000    9.000   16.000
    2    3.000   20.000    7.000   12.000
    3   22.000   17.000   10.000   11.000
    4   13.000    8.000    1.000   23.000
    5    2.000    4.000    6.000   14.000
    6   18.000   21.000   24.000   19.000

   9  & A2; FIELDWIDTH=9; DECIMALS=3

            A2
             1        2        3        4

    1   12.276    8.313    6.392   17.304
    2    9.910   13.615   11.243    8.598
    3   14.515   16.834   13.730   15.012
    4   13.861    5.373    3.681   22.660
    5    7.426    4.232    3.165   11.087
    6   19.119   24.122   19.801   18.258
```

In Example 4.10.1b, the diagonal matrix `Sa` saves the first two singular values, while the first two left singular vectors are stored in the matrix `Ua`. `A2` is a least-squares approximation to `A`, based on $r=2$ singular values (known as an Eckart-Young approximation, of rank 2).

One practical application of the singular value decomposition is to form generalized inverses of matrices. If you use the singular value decomposition you obtain the Moore-Penrose generalized inverse, sometimes called the *pseudo-inverse*, and this is the method used by the `GINVERSE` function.

Example 4.10.1c verifies that the necessary properties of the Moore-Penrose inverse are satisfied. You need to set the `ZDZ` option of `CALCULATE` to `zero` when calculating `Svplus`, the generalized inverse of the diagonal matrix of singular values, in case any of the singular values is zero. The default for `ZDZ` would set the corresponding elements of `Svplus` to be missing (4.1.1).

Example 4.10.1c

```
SVD A; LEFT=Uda; SINGULAR=Sda; RIGHT=Vda
CALCULATE [ZDZ=zero] Svplus = Sda / Sda / Sda
& Aplus = Vda *+ Svplus *+ TRANSPOSE(Uda)
& Aa,Aap = A,Aplus *+ Aplus,A *+ A,Aplus
& CheckAa,CheckAp = MAX(ABS(A,Aplus-Aa,Aap))
" If Aplus is the generalized inverse of A, then
  Aa and Aap should be identical to A and Aplus."
PRINT CheckAa,CheckAp; DECIMALS=3

   CheckAa     CheckAp
     0.000       0.000

CALCULATE Asa,Aspa = A,Aplus *+ Aplus,A
PRINT Asa; FIELDWIDTH=9; DECIMALS=3

                Asa
                  1        2        3        4        5        6

         1    0.398   -0.305    0.113    0.248    0.158    0.218
         2   -0.305    0.845    0.059    0.124    0.083    0.109
         3    0.113    0.059    0.787    0.115   -0.354    0.113
         4    0.248    0.124    0.115    0.762    0.208   -0.219
         5    0.158    0.083   -0.354    0.208    0.409    0.203
         6    0.218    0.109    0.113   -0.219    0.203    0.798

& Aspa; FIELDWIDTH=9; DECIMALS=3

               Aspa
                  1        2        3        4

         1    1.000    0.000    0.000    0.000
         2    0.000    1.000    0.000    0.000
         3    0.000    0.000    1.000    0.000
         4    0.000    0.000    0.000    1.000
```

Singular values and vectors can also be obtained from the `SVALUES`, `LSVECTORS` and `RSVECTORS` functions (which use the same source code within Genstat as `SVD`).

4.10.2 Eigenvalue decompositions: the `FLRV` directive

`FLRV` directive

Forms the values of LRV structures.

Options

`PRINT` = *string tokens*	Printed output required (`roots, vectors`); default `*` i.e. no printing
`NROOTS` = *scalar*	Number of roots or vectors to print; default `*` i.e. print them all
`SMALLEST` = *string token*	Whether to print the smallest roots instead of the largest (`yes`, `no`); default `no`
`TOLERANCE` = *scalar*	Tolerance for detecting zero roots

Parameters

`INMATRIX` = *matrices* or *symmetric matrices*	Matrices whose latent roots and vectors are to be calculated
`LRV` = *LRVs*	LRV to store the latent roots and vectors from each `INMATRIX`
`WMATRIX` = *symmetric matrices*	(Generalized) within-group sums of squares and products matrix used in forming the two-matrix decomposition; if any of these is omitted, it is taken to be the identity matrix, giving the usual spectral decomposition
`ILRV` = *LRVs*	LRV to store the imaginary parts of the latent roots and vectors arising from the decomposition of a non-symmetric matrix

When the `WMATRIX` parameter is unset, `FLRV` solves the eigenvalue problem

$AX = XL$.

In the usual situation A is an n-by-n symmetric matrix, and XLX' provides its eigenvalue (or *spectral*) decomposition. L is a diagonal matrix containing the n latent roots, or eigenvalues, of A ordered such that

$l_1 \geq l_2 \geq ... \geq l_\mathrm{n}$

X is a (square) n-by-n matrix whose columns contain the corresponding latent vectors, or eigenvectors. The matrix X is orthogonal: i.e.

$X'X = XX' = I_n$

The method used for the eigenvalue decomposition first reduces the matrix to tri-diagonal form using Householder transformations (Martin, Reinsch & Wilkinson 1968); this is followed by a QL algorithm for finding the eigenvalues and eigenvectors (Bowdler, Martin, Reinsch & Wilkinson 1968). The decomposition is the basis for several multivariate methods, and is used by several of the procedures in the Genstat Library.

The `INMATRIX` parameter lists the matrices A for which latent roots and vectors are to be calculated. The matrices of latent roots and vectors (L and X) can be saved as the first two elements of the corresponding LRV structure specified by the `LRV` parameter. The third element of the LRV (labelled `'Trace'`) stores the sum of the latent roots, which in this case is also the trace of the original matrix A. Latent roots are often expressed as percentages of the trace. You must declare the LRV structure in advance if you want to save less than the full number of roots; otherwise, it is defined automatically to have n rows.

The three options of `FLRV` control the printing of the results. You use the `PRINT` option to specify whether you want the roots or vectors to be printed. If you request the roots to be printed, the trace will be printed as well. By default nothing is printed. The `NROOTS` option governs how many of the roots and vectors are printed, while the `SMALLEST` option determines whether the largest or smallest roots, and corresponding vectors, are printed.

Example 4.10.2a forms the LRV structure, `Ulrv`, from a matrix of sums of squares and products (4.10.3), and prints the two smallest roots in order of descending magnitude. The trace is printed together with the latent roots, and the latent roots are printed as percentages of the trace.

Example 4.10.2a

```
   2  VARIATE [NVALUE=13] U[1...7]
   3  OPEN 'Harvf.dat'; CHANNEL=2
   4  READ [CHANNEL=2] U[]
```

```
    Identifier   Minimum      Mean   Maximum    Values   Missing
          U[1]     7.910     10.62     12.71        13         0
          U[2]     7.710     10.25     12.15        13         0
          U[3]     8.320     10.46     13.16        13         0
          U[4]     9.190     10.86     13.06        13         0
          U[5]     7.720     10.46     13.08        13         0
          U[6]     8.690     10.53     12.82        13         0
          U[7]     8.810     10.31     11.99        13         0

   5  SSPM [TERMS=U[]] Us; SSP=Ussp
   6  FSSPM Us
   7  FLRV [PRINT=roots,vectors; NROOTS=2; SMALLEST=yes] Ussp; LRV=Ulrv

Spectral decomposition
======================

Latent roots
------------

            6           7
        6.881       1.122

Percentage variation
--------------------

            6           7
         4.25        0.69

Trace
-----

        161.7

Latent vectors
--------------

                         6           7
         U[1]       0.4770      0.4040
         U[2]      -0.4345      0.1617
         U[3]      -0.0525     -0.1007
         U[4]       0.1123     -0.4762
         U[5]       0.1425      0.0629
         U[6]       0.4178      0.5527
         U[7]      -0.6111      0.5141
```

You can save a subset of the latent roots and vectors by supplying an LRV structure with fewer columns than rows. However this saves only the largest roots and the corresponding vectors. You cannot save the smallest roots directly, as the `SMALLEST` option applies only to printing. If you want to save the smallest roots, then you must save the complete set of roots and vectors, and extract the last columns of the matrix, for example using qualified identifiers (4.1.6). These rules are the same as those applied in the directives for multivariate analysis (Part 2 Chapter 6).

You can also set `INMATRIX` to a square, unsymmetric, matrix A. The problem to solve is the *unsymmetric* eigenvalue problem

$$AX = XL.$$

L is again a diagonal matrix of n latent roots (eigenvalues), and X is a square matrix of order n containing the right latent vectors (or eigenvectors) of A. However, the solution may produce some complex latent roots, occurring as complex conjugate pairs, in which case the corresponding latent vectors are also complex conjugate pairs. The `LRV` parameter now saves only the real parts of the latent roots and vectors, and the imaginary parts are saved by the `ILRV` parameter. `ILRV` need not be set, but a warning message is then printed if any complex roots are produced.

If all the latent roots are real, they are sorted into descending order, such that $l_1 \geq l_2 \geq ... l_n$, as in the symmetric case, but if some roots are complex they are ordered such that $|l_1| \geq |l_2| \geq ... \geq |l_n|$. To detect whether a latent root is real, Genstat checks whether imaginary part is close to zero; to allow for numerical imprecision the value is tested against $|l_1|$ multiplied by the valued supplied by the TOLERANCE option, by default 10^{-6}. The values saved by the LRV and ILRV parameters, however, are those generated by the algorithm, so procedures using FLRV may also need to test explicitly for zero roots.

The latent vectors x_i are normalized so that $x_i'x_i=1$, but this is not sufficient to determine them uniquely since they can still be scaled by any (complex) scalar z such that $|z|=1$. The convention adopted in Genstat is to apply an additional scaling such that the largest element of each x_i is real and positive. The latent vectors are guaranteed to be orthogonal only when the matrix A is symmetric.

The algorithm used by FLRV is determined solely by whether the INMATRIX parameter is set to a symmetric matrix structure or to a (square) matrix structure. Symmetric matrices are best stored in a symmetric matrix structure in order to save space, and to use the more efficient symmetric decomposition algorithm. If INMATRIX is set to a matrix A of order n which happens to be symmetric the results should be identical, up to the sign of the latent vectors, apart from small numerical discrepancies of the order of machine precision and dependent on n and the condition number of A. The algorithm used to solve the unsymmetric eigenvalue problem is based on NAG Library subroutine F02EBF. The documentation of this routine should be consulted for a full discussion of the method and accuracy of the results (NAG 1994).

When INMATRIX is set to a square matrix, the WMATRIX parameter is ignored. Similarly, the TOLERANCE option and ILRV parameter are ignored if INMATRIX is set to a symmetric matrix. Percentage variations are printed only if all roots are real.

Example 4.10.2b illustrates the use of FLRV to find the latent roots and vectors of a square matrix A.

Example 4.10.2b

```
MATRIX [ROWS=4;COLUMNS=4] A
READ [PRINT=data,errors] A

 0.35   0.45  -0.14  -0.17
 0.09   0.07  -0.54   0.35
-0.44  -0.33  -0.03   0.17
 0.25  -0.32  -0.13   0.11  :
FLRV [PRINT=roots,vectors] A

Unsymmetric matrix decomposition
================================

Latent roots (real)
-------------------

           1          2          3          4
      0.7995    -0.0994    -0.0994    -0.1007

Latent roots (imaginary)
------------------------

           1          2          3          4
      0.0000     0.4008    -0.4008     0.0000

Trace
-----

      0.5000

Latent vectors (real)
---------------------

                      1          2          3          4
           1     0.6551    -0.1933    -0.1933     0.1253
           2     0.5236     0.2519     0.2519     0.3320
           3    -0.5362     0.0972     0.0972     0.5938
           4      0.0956      0.6760      0.6760      0.7221

Latent vectors (imaginary)
--------------------------

                       1           2           3           4
           1      0.0000      0.2546     -0.2546      0.0000
           2      0.0000     -0.5224      0.5224      0.0000
           3      0.0000     -0.3084      0.3084      0.0000
           4      0.0000      0.0000      0.0000      0.0000
```

When the `WMATRIX` parameter is set, `FLRV` solves the *two-matrix decomposition*

$$AX = WXL$$

A and W are symmetric matrices with the same number of rows, n, and W must be positive semi-definite. L is again a diagonal matrix of size n, and X is a (square) n-by-n matrix. The latent roots, contained in matrix L, are the successive maxima of

$$l = (x'Ax) / (x'Wx)$$

where x is the corresponding column of the matrix X, normalized so that $X'WX=I$.

The method used to solve the two-matrix problem involves two spectral decompositions, each computed as for the one-matrix problem above. The two-matrix decomposition is particularly relevant for canonical variates analysis (see directive `CVA`).

The `WMATRIX` parameter supplies the matrix W, and A is specified by the `INMATRIX` parameter as before. The `LRV` parameter can again be used to save the latent roots and vectors, and the trace. However, the trace is now the trace of $W^{-1}A$.

As an example we take W to be the diagonal of the matrix A. In this case, the solution is equivalent to the spectral decomposition of the correlation matrix derived from A, although the normalization of the latent vectors will be different. Example 4.10.2c shows the equivalence of the two analyses.

Example 4.10.2c

```
  15  CALCULATE Usspcor = CORRMAT(Ussp)
  16  PRINT Usspcor; FIELDWIDTH=7; DECIMALS=3

            Usspcor

           1  1.000
           2  0.215  1.000
           3  0.179  0.113  1.000
           4  0.294  0.439 -0.002  1.000
           5 -0.137  0.100 -0.049  0.345  1.000
           6 -0.754 -0.013 -0.014  0.065  0.062  1.000
           7  0.177 -0.112  0.021  0.419  0.258 -0.359  1.000
                  1      2      3      4      5      6      7

  17  DIAGONALMATRIX Dusp
  18  SYMMETRICMATRIX Sdusp
  19  CALCULATE Sdusp = (Dusp = Ussp)
  20  LRV [ROWS=7; COLUMNS=7] Uclrv,Usclrv
  21  FLRV [PRINT=roots,vectors; NROOTS=4] Usspcor; LRV=Uclrv

Spectral decomposition
======================

Latent roots
------------

           1           2           3           4
       2.114       1.593       1.244       0.936
```

```
Percentage variation
--------------------

        1          2          3          4
    30.20      22.76      17.78      13.37

Trace
-----

    7.000

Latent vectors
--------------

                    1            2            3            4
           1   0.5558      -0.3518       0.1574      -0.1238
           2   0.2649       0.2787       0.6400      -0.2819
           3   0.1227      -0.1055       0.4073       0.8902
           4   0.4249       0.5042       0.1067      -0.0873
           5   0.1510       0.5429      -0.2671       0.1397
           6  -0.4809       0.4648       0.1942       0.1236
           7   0.4138       0.1497      -0.5284       0.2651

  22  & Ussp; LRV=Uclrv; WMATRIX=Sdusp

Two-matrix latent decomposition
===============================

Latent roots
------------

        1          2          3          4
    2.114      1.593      1.244      0.936

Percentage variation
--------------------

        1          2          3          4
    30.20      22.76      17.78      13.37

Trace
-----

    7.000

Latent vectors
--------------

                    1            2            3            4
        U[1]  0.09616     -0.06086      0.02723      0.02142
        U[2]  0.06482      0.06821      0.15664      0.06899
        U[3]  0.02494     -0.02144      0.08279     -0.18093
        U[4]  0.09889      0.11735      0.02484      0.02033
        U[5]  0.02429      0.08736     -0.04298     -0.02248
        U[6] -0.10395      0.10046      0.04198     -0.02672
        U[7]  0.13848      0.05011     -0.17682     -0.08871
```

A similar use of the two-matrix problem is when *W* is obtained from previous samples of the same set of variables as those in *A*.

For a symmetric matrix *A*, you can use `FLRV` to form an inverse of *A* in much the same way as the singular value decomposition. If *A* is singular, this forms the Moore-Penrose inverse (pseudo inverse). Example 4.10.2d follows the lines of the `SVD` example for the generalized inverse of a matrix (4.10.1).

Example 4.10.2d

```
  23  SYMMETRICMATRIX [ROWS=3; VALUES=10,13,17,17,22,29] Smx
  24  LRV [ROWS=3; COLUMNS=3] Lsmx; VECTORS=Vsmx; ROOTS=Rts
  25  FLRV [PRINT=roots,vectors] Smx; LRV=Lsmx

Spectral decomposition
======================

Latent roots
------------

           1            2            3
       55.80         0.20         0.00

Percentage variation
--------------------

           1            2            3
       99.65         0.35         0.00

Trace
-----

       56.00

Latent vectors
--------------

                        1            2            3
           1       0.4233       0.0512      -0.9045
           2       0.5500       0.7788       0.3015
           3       0.7199      -0.6251       0.3015

  26  " The value 1.E-6 is to check for roots which,
 -27    but for numerical round-off, would be zero.
 -28    This might need to be changed in another example. "
  29  CALCULATE [ZDZ=zero] Irts = (Rts > 1.E-6) / Rts
  30  CALCULATE Ismx = Vsmx *+ Irts *+ TRANSPOSE(Vsmx)
  31  PRINT Ismx; FIELDWIDTH=8; DECIMALS=2

                   Ismx
                      1       2       3

           1       0.02    0.21   -0.16
           2       0.21    3.08   -2.46
           3      -0.16   -2.46    1.99
```

The relationship between the singular value decomposition of a rectangular matrix A and the spectral decompositions of $A'A$ and AA' is as follows. If $A = USV'$ is the singular value decomposition for A, then $A'A = VSU'USV' = VS^2V'$ and $AA' = USV'VSU' = US^2U'$, since $U'U = V'V = I$. The rank of matrix A is q and $q \leq \min(m,n)$, which is p in our earlier notation (4.10.1); q corresponds to the number of non-zero singular values, and the diagonal matrix S consists of the q non-zero singular values followed by $(p-q)$ zero values. This shows that the squares of the q singular values of A are equivalent to the non-zero latent roots of the two symmetric matrices, $A'A$ and AA', derived from A. It also shows that the matrices U and V contain the first p latent vectors of AA' and $A'A$, respectively. For further details, see Rao (1973, Chapter 1) or Digby & Kempton (1987, Appendix A.8).

Eigenvalues and vectors can also be obtained from the `EVALUES` and `EVECTORS` functions (which use the same source code within Genstat as `LRV`).

4.10.3 Forming sums of squares and products: the FSSPM directive

FSSPM directive

Forms the values of SSPM structures.

Options

PRINT = *string tokens*	Printed output required (correlations, wmeans, SSPM); default * i.e. no printing
WEIGHTS = *variate* or *symmetric matrix*	Variate of weights for weighted SSP, or symmetric matrix of weights (one row and column for each unit of data); default * i.e. all units with weight one
SEQUENTIAL = *scalar*	Used for sequential formation of SSPMs; a positive value indicates that formation is not yet complete (see READ directive); default * i.e. not sequential

Parameter

SSPMs	Structures to be formed

FSSPM forms the values for the component parts of SSPM structures, based on the information supplied when the SSPM structures were declared (2.7.2). You can use an SSPM as input to the regression directive TERMS (2:3.2.3), or the multivariate directives PCP (2:6.2.1) and CVA (2:6.3.1). The method used to form the SSPM is based on the updating formula for the means and corresponding corrected sums of squares and cross products (Herraman 1968).

FSSPM has one parameter which lists the SSPM structures whose values are to be formed. Genstat takes account of restrictions on any of the variates or factors forming the terms of the SSPM, or on the weights variate or grouping factor if you have specified them. If any of these vectors has a missing value, the corresponding unit is excluded from all the means and all the sums of squares and products. You can also exclude units by setting their weights to zero.

In Example 4.10.3a, units 1, 5 and 7 are omitted. Notice that the wmean setting of the PRINT option is ignored, as the GROUPS option of the SSPM directive has not been set.

Example 4.10.3a

```
   2  VARIATE [NVALUE=10] Va[1...6]
   3  OPEN 'Harvfb.dat'; CHANNEL=2
   4  READ [CHANNEL=2] Va[]

    Identifier   Minimum      Mean   Maximum    Values   Missing
         Va[1]     15.70     36.86     47.10        10         0
         Va[2]     32.30     37.91     55.60        10         0    Skew
         Va[3]     29.40     37.47     53.00        10         0
         Va[4]     26.20     33.66     44.00        10         0
         Va[5]     13.20     38.06     51.90        10         0
         Va[6]     12.70     36.74     54.60        10         0

   5  VARIATE Weight; VALUES=!(0,1,1,1,0,1,0,1,1,1)
   6  SSPM [TERMS=Va[]] Ssva
   7  FSSPM [PRINT=wmean,correlation,sspm; WEIGHT=Weight] Ssva


Degrees of freedom
------------------

Sums of squares:  6
Sums of products: 5
Correlations:     5
```

```
Sums of squares and products
----------------------------

        Va[1]   1       482.54
        Va[2]   2        91.37        88.11
        Va[3]   3       248.40       141.25       559.24
        Va[4]   4       -82.84      -105.96       -75.79       270.99
        Va[5]   5      -305.37       -52.51      -142.92       248.19
        Va[6]   6       122.76       -30.11       248.49        43.17
                             1            2            3            4

        Va[5]   5       983.05
        Va[6]   6      -593.52       799.23
                             5            6


Means
-----

        Va[1]   1        33.54
        Va[2]   2        35.39
        Va[3]   3        38.34
        Va[4]   4        34.67
        Va[5]   5        34.17
        Va[6]   6        34.27


Sum of weights
--------------

        7.000


Correlation matrix
------------------

        Va[1]   1  1.000
        Va[2]   2  0.443  1.000
        Va[3]   3  0.478  0.636  1.000
        Va[4]   4 -0.229 -0.686 -0.195  1.000
        Va[5]   5 -0.443 -0.178 -0.193  0.481  1.000
        Va[6]   6  0.198 -0.113  0.372  0.093 -0.670  1.000
                       1      2      3      4      5      6
```

When you have very many units, you may not be able to store them all at the same time within Genstat. You can then use the `SEQUENTIAL` option of `READ` (3.1.10) to read the data in conveniently sized blocks, and the `SEQUENTIAL` option of `FSSPM` to control the accumulation of the sums of squares and products. The SSPM is updated for each block of data in turn until the end of data is found.

Example 4.10.3b

```
OPEN 'Harv.dat'; CHANNEL=3
SCALAR Sseq; 0
VARIATE [NVALUE=10] V[1...5]
SSPM [TERMS=V[]] Vssp
FOR [NTIMES=999]
   READ [CHANNEL=3; SEQUENTIAL=Sseq] V[]
   FSSPM [SEQUENTIAL=Sseq; PRINT=SSPM] Vssp
   EXIT Sseq <= 0
ENDFOR

    Identifier   Minimum      Mean   Maximum    Values   Missing
          V[1]     8.520     10.13     11.75        10         0
          V[2]     8.910     9.829     10.80        10         0
          V[3]     8.690     10.95     13.08        10         0
          V[4]     7.710     10.01     11.65        10         0
          V[5]     9.290     10.50     12.34        10         0
    Identifier   Minimum      Mean   Maximum    Values   Missing
          V[1]     8.320     10.29     12.71        10         0
          V[2]     7.720     11.00     13.16        10         0
          V[3]     8.930     10.79     12.66        10         0
          V[4]     7.910     11.02     13.06        10         0
          V[5]     8.810     10.90     13.07        10         0

    Identifier   Minimum      Mean   Maximum    Values   Missing
          V[1]     10.67     10.67     10.67        10         9
          V[2]     12.15     12.15     12.15        10         9
          V[3]     10.67     10.67     10.67        10         9
          V[4]     13.06     13.06     13.06        10         9
          V[5]     12.89     12.89     12.89        10         9


Degrees of freedom
------------------

Sums of squares:  20
Sums of products: 19


Sums of squares and products
----------------------------

         V[1]   1      27.75
         V[2]   2       4.72      39.99
         V[3]   3      -4.55      -3.45      34.06
         V[4]   4       4.01      16.81       7.59      55.32
         V[5]   5       6.76      24.19      10.79      16.81      34.71
                           1          2          3          4          5


Means
-----

         V[1]   1        10.23
         V[2]   2        10.50
         V[3]   3        10.86
         V[4]   4        10.64
         V[5]   5        10.80


Number of units used
--------------------

            21
```

Notice that the `PRINT` option has no effect until the last set of values is processed, when `READ` sets the scalar indicator to a negative value (3.1.10).

4.10.4 The QR decomposition

`QRD` directive

Calculates QR decompositions of matrices.

Option

`PRINT` = *string tokens*	Printed output required (`orthogonalmatrix`, `uppertriangularmatrix`); default `*` i.e. no printing

Parameters

`INMATRIX` = *matrices* or *symmetric matrices*	Matrices to be decomposed
`ORTHOGONALMATRIX` = *matrices*	Orthogonal matrix of each decomposition

UPPERTRIANGULARMATRIX = *matrices*
Upper-triangular matrix of each decomposition

QRD uses subroutines F08AEF and F08AFF from the NAG Library to calculate the QR decomposition of a matrix. This is a decomposition of an m by n matrix A into an orthogonal matrix Q (i.e. $Q'Q = I$), and an n by m matrix R, so that $A = Q\,R$.

If $m \geq n$, the top n rows of R are triangular and the lower $m-n$ rows contain zeros. If $m < n$, R is trapezoidal, i.e. it has the form ($R_1 \mid R_2$) where R_1 is an upper triangular matrix and R_2 is a rectangular matrix.

The matrix A to be composed is specified by the INMATRIX parameter, and the matrices Q and R can be saved using the ORTHOGONALMATRIX, and UPPERTRIANGULARMATRIX parameters, respectively.

The PRINT option allows you to print either of the components of the decomposition; by default, nothing is printed.

4.11 Operations on tables

A table is a structure that stores numerical summaries of data that are classified into groups. The TABULATE directive forms tables from a variate (accompanied by factors to define the groups). The tables may contain counts, means, standard errors of means, medians and other quantiles, totals, minima, maxima, variances, standard deviations, skewness or kurtosis coefficients of the observations in each group (4.11.1). You can also use tables to save means, effects and numbers of replications from an analysis of variance (2:4.6.1), or predictions from regression and generalized linear models (2:3.3.4 and 2:3.5.3), or results from stratified surveys (see procedures SVSTRATIFIED and SVTABULATE), or modes (see procedure TABMODE).

As well as these standard types of table, Genstat can form tables involving multiple responses. These occur in surveys as the result of open-ended questions like "Which cities have you visited?". The raw input for a multiple response is often a set of variates or texts (depending on whether the responses were numbers or strings). These can be processed by the FMFACTORS procedure (4.11.8) to form a pointer containing a factor for each possible response code. Summary tables, with one or more dimension corresponding to multiple responses, can then be generated by the MTABULATE procedure (4.11.10). An alternative form of input, free text, finds the responses as keywords within pieces of ordinary text supplied by the respondents. These can be processed, again to form a pointer of factors, using the FFREERESPONSEFACTOR procedure (4.11.9).

You can do numerical calculations on the values in tables, using the CALCULATE directive (4.1.1, 4.1.4 and 4.2.5). You can copy tables into matrices (4.1.3). You can re-form a table to omit or combine levels of any of the classifying factors (4.11.4). You can include margins, or omit them, or recalculate them (4.11.2). You can express the body of a table as percentages of one of its margins (4.11.3). You can also sort the contents of a table to put its margins into ascending or descending order, as required for a Pareto chart (4.11.6).

4.11.1 Tabulation: the TABULATE directive

TABULATE directive
Forms summary tables of variate values.

Options
PRINT = *string tokens* Printed output required (counts, totals, nobservations, means, minima, maxima, variances, quantiles, sds, skewness, kurtosis,

	`semeans`, `seskewness`, `sekurtosis`); default `*` i.e. no printing
`CLASSIFICATION` = *factors*	Factors classifying the tables; default `*` i.e. these are taken from the tables in the parameter lists
`COUNTS` = *table*	Saves a table counting the number of units with each factor combination; default `*`
`SEQUENTIAL` = *scalar*	Used for sequential formation of tables; a positive value indicates that formation is not yet complete (see `READ)`; default `*`
`MARGINS` = *string token*	Whether the tables should be given margins if not already declared (`yes`, `no`); default `no`
`IPRINT` = *string token*	Whether to print the identifier of the table or the identifier of the (associated) variate that was used to form it (`identifier`, `extra`, `associatedidentifier`); default `iden`
`WEIGHTS` = *variate*	Weights to be used in the tabulations; default `*` indicates that all units have weight 1
`PERCENTQUANTILES` = *scalar* or *variate*	Percentage points for which quantiles are required; default 50 (i.e. median)
`OWN` = *scalar* or *variate*	Specifies option settings for the `OWNTAB` subroutine and indicates that this is to supply the data values instead of the variates in the `DATA` list; default `*`
`OWNFACTORS` = *factors*	Factors whose values are to be read by `OWNTAB` (must include the factors of the classification set); default `*`
`OWNVARIATES` = *variates*	Variates whose values are to be read by `OWNTAB` (must include the `DATA` variates); default `*`
`INCHANNEL` = *scalar*	Channel number of the file from which the `OWNTAB` subroutine is to read the data (previously opened by an `OPEN` statement)
`INFILETYPE` = *string token*	Type of the `OWN` data file (`input, unformatted`); default `inpu`

Parameters

`DATA` = *variates*	Data values to be tabulated
`TOTALS` = *tables*	Tables to contain totals
`NOBSERVATIONS` = *tables*	Tables containing the numbers of non-missing values in each cell
`MEANS` = *tables*	Tables of means
`MINIMA` = *tables*	Tables of minimum values in each cell
`MAXIMA` = *tables*	Tables of maximum values in each cell
`VARIANCES` = *tables*	Tables of cell variances
`QUANTILES` = *tables* or *pointers*	Table to contain quantiles at a single `PERCENTQUANTILE` or pointer of tables for several `PERCENTQUANTILEs` (not available for sequential or `OWN` tabulation)
`SDS` = *tables*	Tables of standard deviations
`SKEWNESS` = *tables*	Tables of skewness coefficients
`KURTOSIS` = *tables*	Tables of kurtosis coefficients
`SEMEANS` = *tables*	Tables of standard errors of means
`SESKEWNESS` = *tables*	Tables of standard errors of skewness coefficients

SEKURTOSIS = *tables*	Tables of standard errors of kurtosis coefficients

TABULATE allows you to produce the various types of tabular summary listed in the settings of its PRINT option. The variates whose values are to be summarized are listed with the DATA parameter. If you want to save the summaries in tables, for manipulating or for printing later on, you should list identifiers of the tables in the appropriate parameter list: for example, you would save the totals in a table T by including T in the list for the TOTALS parameter. The other parameters similarly give the other kinds of summary: numbers of non-missing values, means, minima, maxima, variances, quantiles, standard deviations, skewness, kurtosis, (within-cell) standard errors of means, skewness and kurtosis. If you specify less tables in the lists than the number of DATA variates, Genstat produces accumulated summaries. For example, with

```
TABULATE Sales2001,Costs2001,Sales2002,Costs2002;\
  TOTALS=Totalsales,Totalcosts
```

the TOTALS list is recycled. So Totalsales will correspond to Sales2001 and Sales2002, and accumulate the totals from both variates. Similarly Totalcost will contain the totals from the variates Costs2001 and Costs2002. To avoid confusion, however, you are not allowed to specify table lists with differing lengths.

The simplest quantile, and the one produced by default, is the median (50% quantile), but the PERCENTQUANTILE option allows you to request any percentage point (between 0 and 100, of course). Moreover, by specifying a variate as the setting for PERCENTQUANTILE, you can obtain several quantiles at the same time. However, if you then want to save the results the setting of the QUANTILE parameter must be a pointer with length equal to the required number of quantiles, instead of a single table.

If you merely want to print the summaries, you do not usually need to list any tables; you need only specify the PRINT option. The only exception to this is with sequential tabulation, described at the end of this subsection.

The CLASSIFICATION option defines the classifying factors for the tables. This need not be set if at least one of the tables has already been declared (but then all the declared tables must have the same classifying factors). The MARGINS option determines whether or not the tables will have margins, if none have already been declared (and those that have been declared must be either all with margins or all without margins).

Example 4.11.1a concerns goods of two different types dispatched to four different towns. In the print of the data you will notice that the book-keeping has been rather slack. There is one consignment (in line 7) where the type has not been recorded. With such observations, Genstat cannot find out what the group should be because one of the factor values is missing; so they are ascribed to the *unknown cell* associated with the table (2.5). In the declaration in line 10, the scalar that stores this value has been named so that it can be referred to in later calculations. After the tabulation (line 11), table Totdisp stores the total number of items of each type dispatched to each town, and the scalar Udisp summarizes the observations with unknown type or destination.

Example 4.11.1a

```
  2  VARIATE [NVALUES=15] Quantity,Charge
  3  FACTOR [NVALUES=15; LABELS=!T(A,B)] Type
  4  & [LABELS=!T(London,Manchester,Birmingham,Bristol)] Town
  5  READ [PRINT=data,errors] Town,Quantity,Type; FREPRESENTATION=labels
  6      London 10 A  Manchester   5 B  Birmingham  10 B     Bristol 25 A
  7  Manchester 10 *  Birmingham 100 B      London 200 B  Manchester 25 A
  8     Bristol 50 A  Birmingham  25 A     Bristol  25 B      London 25 A
  9      London 50 B  Manchester  25 B      London  50 A  :
 10  TABLE [CLASSIFICATION=Town,Type] Totdisp; UNKNOWN=Udisp
 11  TABULATE Quantity; TOTALS=Totdisp
 12  PRINT Totdisp; DECIMALS=0
```

```
                Totdisp
         Type         A           B
         Town
       London        85         250
   Manchester        25          30
   Birmingham        25         110
      Bristol        75          25

Unknown cell
             Totdisp             10
```

Example 4.11.1b illustrates what happens when a value of the data variate is missing. Variate `Charge` stores the charge to be made for the transport of each consignment, and you will see that three of the values are missing (because these invoices have not yet been prepared). In the tables listed with the parameters, missing data values are ignored. For example, the table `Invoices` is declared automatically by the `NOBSERVATIONS` parameter to hold the number of invoices sent to each destination; it excludes the observations where `Charge` has a missing value. Similarly `Payment` contains the total charge to be paid on behalf of each destination, ignoring the missing values. You can however obtain a count of the numbers of units that would have contributed to each group if no values had been missing: you use the `COUNTS` option if you want to save the table, or put `PRINT=counts` if you want to print it. So table `Nconsign` contains the total number of consignments made to each destination (regardless of whether the corresponding charge is missing or not). The data variates are irrelevant for counts, and so you need not list any if counts are all that you require.

If there are no observations in one of the groups, the corresponding cell will be zero in a table of numbers of observations or counts; in a table of totals, means, minima, maxima, variances, standard deviations, skewness, kurtosis or standard errors of means the cell will contain a missing value.

Example 4.11.1b

```
READ [PRINT=data,errors] Charge
10 20 15 15 * 60 80 30 25 15 25 15 40 * * :
TABULATE [CLASSIFICATION=Town; COUNTS=Nconsign] DATA=Charge; \
  TOTALS=Payment; NOBSERVATIONS=Invoices
PRINT Nconsign,Invoices,Payment; DECIMALS=0,0,2

               Nconsign    Invoices     Payment
         Town
       London         5           4      145.00
   Manchester         4           2       50.00
   Birmingham         3           3       90.00
      Bristol         3           3       65.00
```

Weighted tables can be obtained by setting the `WEIGHT` option to a variate of weights. You can, in general, think of weights as a set of multipliers which are applied to the data before any operations are performed. Thus, for most aspects of weighted tabulation you can replace x by wx and 1 by w (that is, n by Σw) in the standard formulae; see the table below. This is not what happens in the case of variances, standard deviations (which are square roots of the variances) and quantiles, but it is true for the other functions (including counts).

	Unweighted	Weighted
Count	n	$\Sigma\ w$
Total	$\Sigma\ x$	$\Sigma\ wx$
Nobservations	n	$\Sigma\ w$ (x not missing)
Mean	$\Sigma\ x/n$	$\Sigma\ wx\ /\ \Sigma\ w$
Minimum	Min(x)	Min(wx)
Maximum	Max(x)	Max(wx)
Variance	$\Sigma(x - (\Sigma x/n))^2\ /\ n-1$	$\Sigma w(x - (\Sigma wx/\Sigma w))^2\ /\ \Sigma\ w-1$
Skewness	$\Sigma(x - (\Sigma x/n))^3$ $/\ (\ \Sigma(x - (\Sigma x/n))^2\ )^{3/2}$	$\Sigma\ w\ (x - (\Sigma wx/\Sigma w))^3$ $/\ (\ \Sigma\ w\ (x - (\Sigma wx/\Sigma w))^2\ )^{3/2}$
Kurtosis	$\Sigma(x - (\Sigma x/n))^4$ $/\ (\ \Sigma(x - (\Sigma x/n))^2\ )^2 - 3$	$\Sigma\ w\ (x - (\Sigma wx/\Sigma w))^4$ $/\ \Sigma\ w\ (x - (\Sigma wx/\Sigma w))^2\ )^2 - 3$
s.e. skewness	$\surd(\ \{\ 6n \times (n-1)\ \}$ $/\ \{\ (n-2) \times (n+1) \times (n+3)\ \}\ )$	$\surd(\ \{\ 6\Sigma w \times (\Sigma w - 1)\ \}$ $/\ \{\ (\Sigma w - 2) \times (\Sigma w + 1) \times$ $(\Sigma w + 3)\ \}\ )$ (x not missing)
s.e. kurtosis	$\surd(\ \{\ 24 \times n \times (n-1)^2\ \}$ $/\ \{\ (n-2) \times (n-3) \times (n+5) \times$ $(n+3)\ \}\ )$	$\surd(\ \{\ 24 \times \Sigma w \times (\Sigma w - 1)^2\ \}$ $/\ \{\ (\Sigma w - 2) \times (\Sigma w - 3) \times$ $(\Sigma w + 5) \times (\Sigma w + 3)\ \}\ )$ (x not missing)

A quick look at the formula used for the weighted variance (or the standard deviation) or skewness or kurtosis shows that it breaks down for $\Sigma w<1$; in fact it is valid only when the weights are integer values greater than or equal to zero. Similarly, with quantiles the weights are assumed to specify replicated observations; so these must also be non-negative integers. If an invalid weight is found during the calculation of a variance, skewness, kurtosis or quantile a fault will be reported. Temporary tables will be deleted, but named tables may contain partial results. However, non-integer weights are allowed in other contexts. The standard deviation is the square root of the variance, and the standard error of the mean is the standard deviation divided by the square root of the number of observations.

If you have many observations to summarize, there may be insufficient space within Genstat for you to read them all and then form the tables. To cater for such situations, Genstat allows you to process the data in sections, using the `SEQUENTIAL` option of `TABULATE` in conjunction with the `SEQUENTIAL` option of `READ` (3.1.8). After `READ`, the absolute value of the option indicates the number of units that have been read in this particular section; the value is positive during interim sections and negative or zero once the terminator at the end of the data is reached. `TABULATE` will not print any tables until the final section has been processed. If you want to see the intermediate tables, you can include a `PRINT` statement after the `TABULATE` statement. To allow Genstat to keep contact with the working tables in which the results are accumulating, you must save at least one out of the various types of table for every `DATA` variate. Genstat can then link the working tables to this named table during the course of the sequential tabulation, so that the information is not lost between the successive uses of `TABULATE`.

This is illustrated in Example 4.11.1c, which also shows how to use the `IPRINT` option to print the identifier of the variate from which the table was formed, instead of the identifier of the table. Also notice that this time the table formed has a margin (2.5). As there is only one type of table being printed, Genstat has labelled the margin appropriately (as "Mean"). If several types of table were printed, Genstat would label the margins as "Margin". For tables of quantiles, the margin label is either "Median" (for the 50% quantile) or, say, "25%" for the 25% quantile. These labels are associated with the tables for later use, for example by `PRINT`.

The printed table summarizes the amount of excess baggage per person for the passengers on a particular flight. There are 77 passengers. The factors and variates are declared to have length 20, so the data are read in three sections of size 20 and a final section of size 17. The setting of the `SEQUENTIAL` option is the scalar `S`: it has the value 20 for the first three times that the loop is executed, and -17 on the final time. Notice that the variate `Baggage` is given missing values in units 18 to 20 in the final section: the value -17 in `S` tells Genstat that these units are not to be included in the tabulation. The loop construct `FOR-ENDFOR` is described in 5.2.1, and the `EXIT` directive in 5.2.4.

Example 4.11.1c

```
UNITS [NVALUES=20]
FACTOR [LABELS=!T(UK,EEC,other)] National
FACTOR [LABELS=!T(male,female)] Sex
VARIATE Baggage
VARIATE Excess; EXTRA=' baggage per person in Kilograms'; DECIMALS=3
OPEN 'Flight.dat'; CHANNEL=2
SCALAR S
FOR [NTIMES=999]
  READ [CHANNEL=2; SEQUENTIAL=S] Baggage,Sex,National; \
    FREPRESENTATION=labels
  CALCULATE Excess=(Baggage>20)*(Baggage-20)
  TABULATE [PRINT=mean; CLASSIFICATION=Sex,National; SEQUENTIAL=S; \
    MARGINS=yes; IPRINT=associatedidentifier] \
    Excess; NOBSERVATIONS=Ntemp; MEANS=Mtemp
  EXIT S <= 0
ENDFOR

  Identifier   Minimum      Mean   Maximum    Values   Missing
     Baggage     15.00     20.50     28.00        20         0

  Identifier    Values   Missing    Levels
         Sex        20         0         2
    National        20         0         3


  Identifier   Minimum      Mean   Maximum    Values   Missing
     Baggage     17.00     22.45     35.00        20         0


  Identifier    Values   Missing    Levels
         Sex        20         0         2
    National        20         0         3


  Identifier   Minimum      Mean   Maximum    Values   Missing
     Baggage     15.00     20.65     30.00        20         0

  Identifier    Values   Missing    Levels
         Sex        20         0         2
    National        20         0         3


  Identifier   Minimum      Mean   Maximum    Values   Missing
     Baggage     15.00     20.35     28.00        20         3

  Identifier    Values   Missing    Levels
         Sex        20         3         2
    National        20         3         3


                   Excess
```

```
National          UK         EEC       other        Mean
     Sex
    male       1.292       2.364       3.857       2.265
  female       1.200       1.400       1.250       1.250
    Mean       1.256       2.063       2.909       1.896
```

The final five options of `TABULATE` (`OWN`, `OWNFACTORS`, `OWNVARIATES`, `INCHANNEL` and `INFILETYPE`) are appropriate when you have linked your own Fortran subroutine, `G5XZIT`, to Genstat so that you can handle complicated arrangements of data, as may occur for example in hierarchical surveys. This facility is not available in every implementation of Genstat.

G5XZIT is a Fortran subprogram, for you to modify as required, which is called from within `TABULATE` for each unit to be tabulated. It contains switches to tell `TABULATE` when a data error occurs or when all the data have been read. To use it you have to link your own version of Genstat, so that your version of G5XZIT will be used instead of the standard version supplied as part of Genstat.

The subprogram can be as simple or as complicated as you like (or need), provided it obeys a few simple rules. A very simple version, reading two variates and two factors, is supplied with Genstat (see below). This should provide sufficient information for you to write your own version, and link it into your own private version of Genstat.

The `OWN` option should be set to a variate allowing you to communicate between your Genstat code and your G5XZIT subprogram. The `OWNFACTORS` option provides the list of factors to be read by G5XZIT. It must include the classifying factors needed in the current `TABULATE` instruction, but it may contain others as well. The `OWNVARIATES` option should provide a similar list of variates. The `INCHANNEL` option should be set to the Genstat channel number of the data file, as specified in a previous `OPEN` statement or in the Genstat command line. The `INFILETYPE` option specifies whether the data file is character (input) or binary (unformatted).

The documentation of G5XZIT is included with the Fortran and so is not repeated here. The following example shows how `TABULATE` can be used with the standard version of G5XZIT, which is set up simply to read two variates and two factors from a sequential character file. The two variates are read with Fortran format F4.2, which means that their values must be in a field of four characters and will be scaled by 100; the two factors are read with format I4, so the factor values must be integer levels in a field of four characters. Here is a short data file with values in this format.

```
1100 200   1   2
1200 100   1   2
1300 100   2   3
1400 200   2   3
1500 200   1   1
1600 300   2   1
```

Example 4.11.1d shows how `TABULATE` can read these values from a file called `Own.dat` and form a tabular summary of the first variate.

Example 4.11.1d

```
" Declare factors F1 and F2 "
FACTOR [LEVELS=2] F1
& [LEVELS=3] F2
" Open data file containing values for V1, V2, F1 and F2 "
OPEN 'Own.dat'; CHANNEL=3
" Print table of means of variate V1, classified by F1 and F2 "
TABULATE [PRINT=means; CLASSIFICATION=F1,F2; OWN=0; \
  OWNFACTORS=F1,F2; OWNVARIATES=V1,V2; INCHANNEL=3] V1
```

```
              Mean
F2               1          2          3
F1
 1           15.00      11.50          *
 2           16.00          *      13.50
```

TABULATE allows only one classification set to be used at a time. If the data set is complicated enough to require G5XZIT, then several tabulations with different classifying sets are likely to be needed. Rather than have a separate branch in G5XZIT for each tabulation, you can put all the factors and all the variates that you will need into the settings of the OWNFACTORS and OWNVARIATES options, and leave TABULATE to extract the ones it needs each time. If you have several TABULATE statements as suggested, you will have to close the data file and re-open it between them.

4.11.2 Forming margins of tables: the MARGIN directive

MARGIN directive

Forms and calculates marginal values for tables.

Option

CLASSIFICATION = *factors*	Factors classifying the margins to be formed; default * requests all margins to be formed

Parameters

OLDTABLE = *tables*	Tables from which the margins are to be taken or calculated
NEWTABLE = *tables*	New tables formed with margins
METHOD = *string tokens*	Way in which the margins are to be formed for each table (totals, means, minima, maxima, variances, medians, deletion, or a null string to indicate that the marginal values are all to be set to the missing value); default tota

You can use MARGIN to extend a table to contain marginal values, or to change the marginal values of a table that already has margins, or to delete the margins from a table. The tables whose margins are to be changed are specified by the OLDTABLE parameter. If you specify only this parameter, the new values replace those of the original tables. For example, in 2.5, the statement

```
MARGIN Classnum,Schoolnm
```

formed margins of totals over all the classifying factors for the tables, Classnum and Schoolnm; the new values, including the margins, replaced the original values of Classnum and Schoolnm.

However, if you want to retain the original values, you can specify new tables to contain the amended values, using the NEWTABLE list. These tables will be declared automatically, if you have not declared them already.

Example 4.11.2a creates the new tables, Classt and Schoolt, with margins of totals, using the values in the tables Classnum and Schoolnm.

Example 4.11.2a

```
  2  FACTOR [LABELS=!T(boy,girl)] Sex
  3  FACTOR [LEVELS=5] Class
  4  FACTOR [LEVELS=2] School
```

```
TABLE [CLASSIFICATION=Class,Sex; \
  VALUES=15,17,29,31,34,30,33,35,28,27] Classnum
TABLE [CLASSIFICATION=School,Class,Sex; VALUES=15,17,29,31,34, \
  30,33,35,28,27,18,16,33,31,35,36,34,33,31,32] Schoolnm
MARGIN Classnum,Schoolnm; NEWTABLE=Classt,Schoolt
PRINT Classt,Schoolt; DECIMALS=0

             Classt
     Sex        boy        girl      Margin
   Class
       1         15          17          32
       2         29          31          60
       3         34          30          64
       4         33          35          68
       5         28          27          55
  Margin        139         140         279


                          Schoolt
                   Sex        boy        girl      Margin
  School         Class
       1             1         15          17          32
                     2         29          31          60
                     3         34          30          64
                     4         33          35          68
                     5         28          27          55
                Margin        139         140         279
       2             1         18          16          34
                     2         33          31          64
                     3         35          36          71
                     4         34          33          67
                     5         31          32          63
                Margin        151         148         299
  Margin             1         33          33          66
                     2         62          62         124
                     3         69          66         135
                     4         67          68         135
                     5         59          59         118
                Margin        290         288         578
```

You can form other types of margin by setting the `METHOD` parameter. The next example forms the tables `Classno` and `Schoolno` with margins of means and maxima respectively.

Example 4.11.2b

```
MARGIN Classnum,Schoolnm; NEWTABLE=Classno,Schoolno; \
  METHOD=means,maxima : PRINT Classno,Schoolno; DECIMALS=0


            Classno
     Sex        boy        girl      Margin
   Class
       1         15          17          16
       2         29          31          30
       3         34          30          32
       4         33          35          34
       5         28          27          28
  Margin         28          28          28


                         Schoolno
                   Sex        boy        girl      Margin
  School         Class
       1             1         15          17          17
                     2         29          31          31
                     3         34          30          34
                     4         33          35          35
                     5         28          27          28
                Margin         34          35          35
       2             1         18          16          18
```

```
                     2          33          31          33
                     3          35          36          36
                     4          34          33          34
                     5          31          32          32
                Margin          35          36          36
    Margin           1          18          17          18
                     2          33          31          33
                     3          35          36          36
                     4          34          35          35
                     5          31          32          32
                Margin          35          36          36
```

All the examples so far have been of adding margins. But you can delete them too: if you set `METHOD=deletion`, all the margins of the tables are deleted but the body of the table is retained.

The `CLASSIFICATION` option specifies the list of factors for which you want to form marginal values. Example 4.11.2c forms a margin of totals for the factor `Class` in the table `Classnum`.

Example 4.11.2c

```
  13  MARGIN [CLASSIFICATION=Sex] Classnum
  14  PRINT Classnum; DECIMALS=0

              Classnum
      Sex          boy        girl      Margin
    Class
        1           15          17          32
        2           29          31          60
        3           34          30          64
        4           33          35          68
        5           28          27          55
   Margin            *           *           *
```

Genstat puts missing values in the margins that are excluded if the `METHOD` parameter is set to maxima or minima; for other settings of `METHOD`, Genstat puts in zeroes.

The classifying sets for each table can be different, but all the factors listed by the `CLASSIFICATION` option must be in the classifying sets of each `OLDTABLE`. So, for example,

```
MARGIN [CLASSIFICATION=Sex,School] Classnum, Schoolnm
```

would fail because the factor `School` is not in the classifying set of `Classnum`.

4.11.3 Forming tables of percentages: the PERCENT and T%CONTROL procedures

PERCENT procedure

Expresses the body of a table as percentages of one of its margins (R.W. Payne).

Options

`CLASSIFICATION` = *factors*	Factors classifying the margin over which the percentages are to be calculated
`METHOD` = *string token*	Method to use to calculate the margin if not already present (`totals`, `means`, `minima`, `maxima`, `variances`, `medians`); default `tota`
`HUNDRED` = *string token*	Whether to put 100% values into the margin instead of the original values (`no`, `yes`); default `no`

Parameters

`OLDTABLE` = *tables*	Tables containing the original values

NEWTABLE = *tables*	Tables to store the percentage values; if any of these is unset, the new values replace those in the original table

The PERCENT procedure allows you to express the body of a table as percentages of the values in one of its margins. The table is specified using the OLDTABLE parameter. A table to store the new values can be specified using the NEWTABLE parameter, otherwise these replace the values of the original table. The margin is indicated by listing the factors that define it using the CLASSIFICATION option; the default is the final margin (the grand total, or grand mean &c). If the original table has no margins, option METHOD defines how these are to be calculated (totals, means, minima, maxima, variances, medians); the default is to form margins of totals. The values originally in the margin will be left unchanged. If you would prefer these to be replaced by values of 100%, you should set option HUNDRED=yes.

In Example 4.11.3a the contents of the table Totdisp, formed in Example 4.11.1a, are expressed as percentages of the overall margin − which, as option HUNDRED is not set, is left with its original value 625. The unknown cell is not included in the calculations, and option setting PUNKNOWN=never in PRINT suppresses it from being printed.

Example 4.11.3a

```
  18  PERCENT Totdisp; NEWTABLE=Totdisp%
  19  PRINT [PUNKNOWN=never] Totdisp%; DECIMALS=2

                  Totdisp%
          Type           A           B       Total
          Town
        London       13.60       40.00       53.60
    Manchester        4.00        4.80        8.80
    Birmingham        4.00       17.60       21.60
       Bristol       12.00        4.00       16.00
         Total       33.60       66.40      625.00
```

T%CONTROL procedure

Expresses tables as percentages of control cells (R.W. Payne).

Option

PRINT = *string token*	Controls printed output (percentages); default perc

Parameters

OLDTABLE = *tables*	Tables containing the original values
NEWTABLE = *tables*	Tables to store the percentage values
FACTOR = *factors* or *pointers*	Factor, or pointer of factors, with control levels
CONTROL = *scalars*, *vaiates*, *texts* or *pointers*	Identifies the control level or levels of each FACTOR (if more than one is specified for a factor, their mean is used); default uses the reference level

T%CONTROL allows you to express the body of a table as percentages of the values of "control" levels of one or more of its classifying factors. These controls might be standard or check varieties in a variety trial, or placebo treatments in a medical trial, or zero levels of fertilizers in an agricultural field experiment, etc.

You supply the table using the OLDTABLE parameter. You can save a new table containing the percentages using the NEWTABLE parameter. The factors containing the control levels are specified by the FACTOR parameter; if there are several you must put them into a pointer. The CONTROL parameter identifies the control levels of each factor. Usually the factor will have a single control, specified either by giving its level (in a scalar) or its label (in a string or single-valued text). Alternatively, you can define several controls, by specifying a variate (of levels) or a multi-valued text (of labels); T%CONTROL then takes means over the control levels. Again, if there are several factors, you must put the corresponding CONTROL settings into a pointer. If CONTROL is unset or missing for any factor, T%CONTROL uses its reference level.

Not all the factors in the table need to have control levels. Suppose, for example, we have a 2-way table with factors A and B where the first level of A (a_1) is the control. Then the cell (a_i, b_j) will be given as a percentage of the cell (a_1, b_j).

By default T%CONTROL prints the table of percentages, but you can set option PRINT=* to suppress this.

Example 4.11.3b the contents of the table Totdisp, formed in Example 4.11.1a, are as percentages of amounts dispatched to London.

Example 4.11.3b

```
  20  T%CONTROL Totdisp; FACTOR=Town; CONTROL='London'

         Type          A           B
         Town
       London     100.00      100.00
   Manchester      29.41       12.00
   Birmingham      29.41       44.00
      Bristol      88.24       10.00


expressed as percentages of Town London.
```

4.11.4 Combining or omitting slices of tables and matrices: the COMBINE directive

COMBINE directive

Combines or omits "slices" of a multi-way data structure (table, matrix or variate).

Options

OLDSTRUCTURE = *identifier*	Structure whose values are to be combined; no default i.e. this option must be set
NEWSTRUCTURE = *identifier*	Structure to contain the combined values; no default i.e. this option must be set

Parameters

OLDDIMENSION = *factors* or *scalars*	Dimension number or factor indicating a dimension of the OLDSTRUCTURE
NEWDIMENSION = *factors* or *scalars*	Dimension number or factor indicating the corresponding dimension of the NEWSTRUCTURE; this can be omitted if the dimensions are in numerical order, while zero settings (each in conjunction with a single OLDPOSITION) allows a slice of an old table to be mapped into a new table with fewer dimensions
OLDPOSITIONS = *pointers*, *texts*, *variates* or *scalars*	These define positions in each OLDDIMENSION: pointers are appropriate for matrices whose rows or columns are indexed by a pointer; texts are for matrices indexed by a

	text, variates with a textual labels vector, or tables whose `OLDDIMENSION` factor has labels; and variates either refer to levels of table factors or numerical labels of matrices or variates, if these are present, otherwise they give the (ordinal) number of the position. If omitted, the positions are assumed to be in (ordinal) numerical order. Margins of tables are indicated by missing values
`NEWPOSITIONS` = *pointers*, *texts*, *variates* or *scalars*	These define positions in each `NEWDIMENSION`, specified similarly to `OLDPOSITIONS`; these indicate where the values from the corresponding `OLDDIMENSION` positions are to be entered (or added to any already entered there)
`WEIGHTS` = *variates*	Define weights by which the values from each `OLDDIMENSION` coordinate are to be multiplied before they are entered in the `NEWDIMENSION`

Sometimes you may wish to reclassify a table to have factors different from those that you used in its declaration. `COMBINE` allows you to omit or to combine levels of the classifying factors. Furthermore, if you want to take just one level of a factor, you can copy the values into a table with one less dimensions.

You specify the original table using the `OLDSTRUCTURE` option, and a table to contain the reclassified values using the `NEWSTRUCTURE` option; if you have not already declared the new table, it will be declared implicitly. You must specify both of these options.

You can modify several of the classifying factors at a time. You list the factors of the original table with the `OLDDIMENSION` parameter, and the equivalent factors of the new table with `NEWDIMENSION`. An alternative way of doing this is to give a dimension number, specifying the position of the factor in the classifying set of the table (2.5); for the `NEWDIMENSION` list, this requires that you have already declared the new table. You can even omit the list of dimensions if they would be in ascending numerical order.

In Example 4.11.4a, the table `Sales` contains the number of items of some product sold by a retailer with shops in nine towns, in the years 1979 to 1984. Lines 21 to 24 form a table `Csales` in which the sales are classified by the country where the sale was made, instead of the town; so there is one `OLDDIMENSION`, the factor `Town`, and a corresponding `NEWDIMENSION`, `Country`.

Example 4.11.4a

```
   2  TEXT [VALUES= Aberdeen,Birmingham,Cardiff,Dundee,Edinburgh, \
   3    Liverpool,Manchester,Sheffield,Swansea] Townname
   4  VARIATE [VALUES=1979,1980,1981,1982,1983,1984] Yearnum
   5  FACTOR  [LABELS=Townname] Town
   6  FACTOR  [LEVELS=Yearnum]  Year; DECIMALS=0
   7  TABLE [CLASSIFICATION=Town,Year] Sales
   8  READ Sales

    Identifier   Minimum      Mean   Maximum     Values   Missing
         Sales     343.0     676.3      1158         54         0

  18  PRINT Sales; FIELDWIDTH=8; DECIMALS=0

                Sales

         Year     1979    1980    1981    1982    1983    1984
         Town
    Aberdeen       608     635     672     692     685     723
```

```
Birmingham      618      601      784      720      863      921
   Cardiff      757      743      785      816      783      737
    Dundee      343      391      358      366      418      470
 Edinburgh      714      751      710      763      788      830
 Liverpool      816      859      820      938     1007     1158
Manchester      662      632      758      721      893      837
 Sheffield      531      569      615      624      607      593
   Swansea      416      461      478      462      497      520

 19  FACTOR [LABELS=!T(England,Wales,Scotland)] Country
 20  " Form a table Csales, classified by country instead of town."
 21  COMBINE [OLDSTRUCTURE=Sales; NEWSTRUCTURE=Csales] \
 22    OLDDIMENSION=Town; NEWDIMENSION=Country; \
 23    OLDPOSITIONS=!(2,6,7,8,1,4,5,3,9); \
 24    NEWPOSITIONS=!T(4(England),3(Scotland),2(Wales))
 25  PRINT Csales; FIELDWIDTH=8; DECIMALS=0

              Csales

      Year     1979     1980     1981     1982     1983     1984
   Country
   England     2627     2661     2977     3003     3370     3509
     Wales     1173     1204     1263     1278     1280     1257
  Scotland     1665     1777     1740     1821     1891     2023
```

Each of the levels of `Country` is a combination of several levels of `Town`. You use the `OLDPOSITIONS` and `NEWPOSITIONS` parameters to specify how this combining is to be done. These parameters specify a pair of vectors for each pair of old and new dimensions, listing positions within the old dimension and the corresponding positions to which they are mapped in the new dimension. The positions can be defined in terms of either the levels or the labels of the factor that classifies the dimension. In the example, the vector for the old dimension `Town` is an unnamed variate `!(2,6,7,8,1,4,5,3,9)` whose values refer to the levels (1 to 9); the vector for `Country` is an unnamed text `!T(4(England),3(Scotland),2(Wales))` whose values are labels of `Country`. The correspondence between the two sets of values is:

Town level	**Town label**	**Country label**	**Country level**
2	Birmingham	England	1
6	Liverpool	England	1
7	Manchester	England	1
8	Sheffield	England	1
1	Aberdeen	Scotland	2
4	Dundee	Scotland	2
5	Edinburgh	Scotland	2
3	Cardiff	Wales	3
9	Swansea	Wales	3

Thus, as you can see, the values in the original table for the English towns (Birmingham, Liverpool, Manchester and Sheffield) are allocated to Country England in the new table, the Scottish towns (Aberdeen, Dundee and Edinburgh) are allocated to Scotland, and Cardiff and Swansea are allocated to Wales.

If you omit the vector for one of the dimensions, it is assumed to contain each value once only, taken in the order in which they occur in the levels vector of the factor. Thus the `OLDPOSITIONS` variate could be omitted in

```
COMBINE[OLDSTRUCTURE=Sales; NEWSTRUCTURE=Csales] \
  OLDDIMENSION=Town; NEWDIMENSION=Country; \
  OLDPOSITIONS=!(1...9); NEWPOSITIONS= \
    !T(Scotland,England,Wales,2(Scotland),3(England),Wales)
```

You indicate a margin of the table by a missing value in a variate, or by a null string in a text.

Values in the original table can be allocated to more than one place. Also, as we have mentioned already, you can modify more than one dimension at a time. In Example 4.11.4b, the `Years` dimension is modified as well as the `Town` dimension: years 1979 and 1980 are omitted, while the other years are allocated to two summary lines as well as to themselves in the new dimension `Ysummary`. Thus the new table `Salesum` has lines giving sales for the individual years, interspersed with bi-annual totals. Note that the interspersing of the summary lines is ensured by the order in which the `FACTOR` declaration specifies the labels of the factor `Yearsum`.

Example 4.11.4b

```
" Form a table classified by country and year,
  including biannual totals."
FACTOR [LABELS=!T('1981','1982','1981-2','1983','1984','1983-4')]\
  Yearsum
TABLE  [CLASSIFICATION=Yearsum,Country] Salesum
COMBINE [OLDSTRUCTURE=Sales; NEWSTRUCTURE=Salesum] \
  OLDDIMENSION=Town,Year; NEWDIMENSION=Country,Yearsum; \
  OLDPOSITIONS=!(2,6,7,8,1,4,5,3,9),!V((1981...1984)2); \
  NEWPOSITIONS=!T(4(England),3(Scotland),2(Wales)), \
  !T('1981','1982','1983','1984',2('1981-2','1983-4'))
PRINT Salesum; FIELDWIDTH=8; DECIMALS=0

        Salesum

Country England   Wales Scotland
Yearsum
   1981    2977    1263    1740
   1982    3003    1278    1821
 1981-2    5980    2541    3561
   1983    3370    1280    1891
   1984    3509    1257    2023
 1983-4    6879    2537    3914
```

The final use of the sales data shows how to extract a single slice of a table into a table with fewer dimensions. In Example 4.11.4c, the `OLDPOSITIONS` parameter specifies a single level `England` of the `OLDDIMENSION` `Country`, the `NEWDIMENSION` is set to 0. The new table is thus classified by only the factor `Yearsum`, and contains information about sales in England.

Example 4.11.4c

```
COMBINE [OLDSTRUCTURE=Csales; NEWSTRUCTURE=Esales] \
  OLDDIMENSION=Country; NEWDIMENSION=0; OLDPOSITIONS='England'
PRINT Esales

             Esales
   Year
   1979        2627
   1980        2661
   1981        2977
   1982        3003
   1983        3370
   1984        3509
```

In parallel with the vectors of positions, you can also specify a variate of weights by which the values are multiplied before being entered into the new table. Thus, for example, forming summary lines of means instead of totals would require an extra parameter list

```
WEIGHTS=*,!(1,1,1,1,0.5,0.5,0.5,0.5)
```

Although the main way in which you will use `COMBINE` is likely to be for tables, you can also use it on rectangular matrices and even variates. For these, the dimensions can only be numbers: number 1 refers to the rows of a matrix, and 2 to the columns; number 1 refers to the rows (or units) of a variate. The position vectors refer to the labels vectors of matrices (2.4.1), which can be variates, texts or pointers; or they refer to the unit labels of a variate (2.3.1), which can be held in either a variate or a text. If a dimension has no labels vector, you use a variate to specify its positions; then each value of the variate gives the number of a row, column or unit. You can do the same also if the labels vector is something other than a variate: that is, a text or a pointer.

4.11.5 Inserting a table into another table: the `TABINSERT` procedure

`TABINSERT` procedure

Inserts the contents of a sub-table into a table (R.W. Payne).

Options

`OLDTABLE` = *tables*	Table containing the original values
`SUBTABLE` = *tables*	Sub-table to insert into the original table
`NEWTABLE` = *tables*	Tables to store the new values; if this is not set, these replace those in the original table

Parameters

`OLDFACTOR` = *factors*	Factors classifying the dimensions of the old table that are smaller in the sub-table
`SUBFACTOR` = *factors*	Specifies the factors classifying the corresponding dimensions of the sub-table
`FREPRESENTATION` = *string token*	How to match the values of each `OLDFACTOR` and `SUBFACTOR` (`levels`, `labels`); default `leve`

`TABINSERT` allows you to replace values in a table by those in a sub-table. It can also be used to insert values into the margins of a table. The original table and the sub-table are specified by the `OLDTABLE` and `SUBTABLE` options, respectively. You can use the `OLDTABLE` option to a specify a table to store the modified table values. If this is not set, they replace those in the original table.

The sub-table will usually have the same number of classifying factors as the original table. Some may be in common (and these can be ignored). Pairs of factors that differ are specified by the `OLDFACTOR` and `SUBFACTOR` parameters. The `FREPRESENTATION` indicates whether the factors are to be matched by their levels (default) or their labels. The idea is that the levels (or labels) of the `SUBFACTOR` are a subset of those of the `OLDFACTOR`, indicating where the values of the sub-table are to be inserted. If you omit some factors of the original table from both the sub-table and the `OLDFACTOR` list, the values of the sub-table are inserted into their margins in the modified table.

If both tables have margins, those in the sub-table will be transferred as well as those in the body of the table. If you want to omit the marginal values, you should remove the margins from the sub-table, using the `MARGIN` directive with parameter `METHOD=deletion`; see 4.11.2. You can also use `MARGIN` to recalculate the margins in the new table, if they are no longer valid after the values in the sub-table have been inserted.

Example 4.11.5 considers a similar situation to that in Example 4.11.1, except that here we want to include some additional summaries which will be calculated independently in Scotland. The first 12 lines form totals for the English data, as in Example 4.11.1a.

Notice, though, that the factor `Town` now has two additional levels for the Scottish towns Edinburgh and Glasgow. The information for these towns is provided, already summarized, in lines 13-16. The table `Scotsdisp` is classified by the factor `Type` (like the table `UKdisp` formed in lines 11 and 12) and the factor `Scotstown` with levels Edinburgh and Glasgow that are a subset of the levels of the factor `Town`.

TABINSERT can then be used, in lines 18-19, to insert the Scottish information into the UK table. The OLDTABLE is UKdisp, and the SUBTABLE is Scotsdisp. NEWTABLE is not specified, so the Scottish summaries are inserted into UKdisp itself. The OLDFACTOR is Town, and the SUBFACTOR is Scotstown.

Example 4.11.5

```
   2  " tabulate the English information "
   3  VARIATE [NVALUES=15] Quantity,Charge
   4  FACTOR  [NVALUES=15; LABELS=!T(A,B)] Type
   5  & [LABELS=!T(London,Manchester,Birmingham,Bristol,Edinburgh,Glasgow)] Town
   6  READ [PRINT=data,errors] Town,Quantity,Type; FREPRESENTATION=labels

   7      London 10 A  Manchester   5 B  Birmingham  10 B     Bristol 25 A
   8  Manchester 10 *  Birmingham 100 B      London 200 B  Manchester 25 A
   9     Bristol 50 A  Birmingham  25 A     Bristol  25 B      London 25 A
  10      London 50 B  Manchester  25 B      London  50 A  :
  11  TABLE     [CLASSIFICATION=Town,Type] UKdisp
  12  TABULATE  [PRINT=totals] Quantity; TOTALS=UKdisp

                  UKdisp
        Type           A           B
        Town
      London       85.00      250.00
  Manchester       25.00       30.00
  Birmingham       25.00      110.00
     Bristol       75.00       25.00
   Edinburgh           *           *
     Glasgow           *           *

Unknown cell
              UKdisp           10.00

  13  " information already summarized from Scotland "
  14  FACTOR    [LABELS=!T(Edinburgh,Glasgow)] Scotstown
  15  TABLE     [CLASSIFICATION=Scotstown,Type; VALUES=20,40,30,120] Scotsdisp
  16  PRINT     Scotsdisp

               Scotsdisp
        Type           A           B
   Scotstown
   Edinburgh       20.00       40.00
     Glasgow       30.00      120.00


  17  " insert the Scottish information into the UK table "
  18  TABINSERT [OLDTABLE=UKdisp; SUBTABLE=Scotsdisp] Town; SUBFACTOR=Scotstown;\
  19            FREPRESENTATION=labels
  20  PRINT     UKdisp

                  UKdisp
        Type           A           B
        Town
      London       85.00      250.00
  Manchester       25.00       30.00
  Birmingham       25.00      110.00
     Bristol       75.00       25.00
   Edinburgh       20.00       40.00
     Glasgow       30.00      120.00

Unknown cell
              UKdisp           10.00
```

4.11.6 Forming a Pareto chart: the **TABSORT** procedure

TABSORT procedure

Sorts tables so their margins are in ascending or descending order (R.W. Payne).

Options

PRINT = *string tokens*	Controls output (tables, histograms); default * i.e. none
DIRECTION = *string token*	Direction of sorting (ascending, descending); default asce
METHOD = *string token*	Method to use to construct a marginal table for the sorting of a factor when there is no one-way table classified by the factor in the TABLE list, and the first table in the TABLE list classified by the factor has no margins (totals, means, minima, maxima, variances, medians); default tota
FACTORS = *pointer*	Specifies or saves a list of classifying factors of the tables in the TABLE list
NEWFACTORS = *pointer*	Specifies or saves a list of classifying factors of the new tables, corresponding to those in the FACTORS pointer
EXCLUDE = *pointer*	Factors to exclude from sorting
NBEST = *string tokens*	Number of (best) levels to include from each sorted factor; default * i.e. all of them

Parameters

TABLE = *tables*	Tables to be sorted
NEWTABLE= *tables*	Allows the new sorted tables to be saved
TITLE = *texts*	Title to be used when displaying each table
FIELDWIDTH = *scalars*	Field width for printing each table
DECIMALS = *scalars*	Decimal places for each table

The TABSORT procedure sorts tables so that their margins are in a specified order. With a multi-way table, for example, this may help in interpreting an interaction from an analysis of variance. With a one-way table, it allows the cells to be displayed in ascending order, as in a Pareto chart.

Example 4.11.6a continues the example concerning goods of two different types dispatched to four different towns, started in Section 4.11.1 and continued in Section 4.11.3. The table Invoices (formed in Example 4.11.1b) is sorted into ascending order and printed.

Example 4.11.6a

```
  21  TABSORT [PRINT=tables; DIRECTION=descending] Invoices; DECIMALS=0

 sorted['Town']
        London          4
    Birmingham          3
       Bristol          3
    Manchester          2
```

The original tables are supplied by the TABLE parameter, and the NEWTABLE parameter can be used to save the sorted tables.

If you want to specify your own ordering, the FACTORS and NEWFACTORS options can be set to pre-defined pointers of factors indicating the ways in which each dimension of the tables is to be sorted: FACTORS contains factors from the classifying sets of the original tables, and NEWFACTORS contains the corresponding factors for the new tables (with the levels in the new order).

Alternatively, as in Example 4.11.6a, you can let TABSORT define the ordering. For each factor classifying the original tables, the ordering is obtained using a one-way table for that factor. This may be available amongst the list of original tables (specified by the TABLE parameter). If not, TABSORT finds the first table in the list with the factor in its classifying set. If the table has margins, then TABSORT will extract the appropriate one-way margin. Otherwise, it first constructs the margins using the MARGIN directive; the METHOD option then defines how the margin is formed (using means, medians and so on). Having obtained a suitable one-way table, TABSORT forms a new factor whose levels are in the order that will arrange the entries of the table in either ascending or descending order according to the setting of the DIRECTION option (default ascending). The FACTORS and NEWFACTORS options can be used to save pointers containing the factors and reordered factors for future use. Note also, that even if you do not want to use the factors in future, you can use the pointers to specify identifiers for the new factors to be used when the tables are printed. The EXCLUDE option can be set to a pointer containing factors that are not to be re-ordered automatically, but should be left unchanged.

The NBEST option specifies the number of levels to include from each sorted factor. So, setting NBEST=5 would take only the first five levels in the sorted order. This may be useful if you have a large table, and want to show only the best part of the table (as defined by the sorting of the margins). This default is to include all of the levels.

The PRINT option controls the output produced by TABSORT. The setting tables prints the tables (see Example 4.11.6a). The setting histograms, causes any one-way tables to be plotted by the DHISTOGRAM directive, and any two-way tables to be plotted by D3HISTOGRAM.

The TITLE parameter allows you to supply a title to be used in the display of each table. The FIELDWIDTH parameter specifies field widths, and the DECIMALS parameter specified numbers of decimal places

Example 4.11.6b sorts the two-way table Totdisp% from Example 4.11.3 to put the margins into ascending order. The new classifying factors (in the pointer Newclassification) are SortedTowns and SortedTypes, and the new table is SortedTotals.

Example 4.11.6b

```
  22  POINTER [VALUES=Town,Type] Classification
  23  POINTER [VALUES=SortedTowns,SortedTypes] Newclassification
  24  TABSORT [DIRECTION=descending; FACTORS=Classification;\
  25          NEWFACTORS=Newclassification] Totdisp%; NEWTABLE=Sorted%Totals
  26  PRINT   Sorted%Totals

            Sorted%Totals

  SortedTypes            B            A       Margin
  SortedTowns
       London        40.00        13.60        53.60
   Birmingham        17.60         4.00        21.60
      Bristol         4.00        12.00        16.00
   Manchester         4.80         4.00         8.80

       Margin        66.40        33.60       625.00
```

4.11.7 Plots of tables: the DTABLE procedure

DTABLE procedure

Plots tables (R.W. Payne).

Options

GRAPHICS = *string token*	Type of graph (highresolution, lineprinter); default high
METHOD = *string token*	What to plot (points, linesandpoints, onlylines, data, barchart, splines); default poin
XFREPRESENTATION = *string token*	How to label the *x*-axis (levels, labels); default labels uses the XFACTOR labels, if available
DFSPLINE = *scalar*	Number of degrees of freedom to use when METHOD=splines
YTRANSFORM = *string tokens*	Transformed scale for additional axis marks and labels to be plotted on the right-hand side of the y-axis (identity, log, log10, logit, probit, cloglog, square, exp, exp10, ilogit, iprobit, icloglog, root); default iden i.e. none
PENYTRANSFORM = *scalar*	Pen to use to plot the transformed axis marks and labels; default * selects a pen, and defines its properties, automatically
KEYMETHOD = *string token*	What to use for the key descriptions when GROUPS specifies more than one factor (labels, namesandlabels); default name
PLOTTITLEMETHOD = *string token*	What to use for the titles of the plots when TRELLISGROUPS specifies more than one factor (labels, namesandlabels); default name
PAGETITLEMETHOD = *string token*	What to use for the titles of the pages when PAGEGROUPS specifies more than one factor (labels, namesandlabels); default name
USEAXES = *string token*	Which aspects of the current axis definitions of window 1 to use (none, limits, marks, mpositions, nsubticks,); default none

Parameters

TABLE = *tables*	Tables to plot
DATA = *variates*	Data values to plot with each table when METHOD=data
XFACTOR = *factors*	Factor providing the *x*-values for the plot of each table
GROUPS = *factors* or *pointers*	Factor or factors identifying the different lines from a multi-way table
TRELLISGROUPS = *factors* or *pointers*	Factor or factors specifying the different plots of a trellis plot of a multi-way table
PAGEGROUPS = *factors* or *pointers*	Factor or factors specifying plots to be displayed on different pages
BAR = *scalars*, *tables* or *pointers*	Scalar defining the length of error bar to be plotted to indicate the overall (or average) variability of the values in each table, or table defining the variability of each individual table value, or pointer containing either two scalars or two tables defining the upper and lower

	positions of the error bar(s)
NEWXLEVELS = *variates*	Values to be used for XFACTOR instead of its existing levels
TITLE = *texts*	Title for the graph; default uses the identifier of the TABLE
YTITLE = *texts*	Title for the y-axis; default ' '
XTITLE = *texts*	Title for the x-axis; default is to use the identifier of the XFACTOR
BARDESCRIPTION = *texts*	Descriptions for the bars
PENS = *variates*	Defines the pen to use to plot the points and/or line for each group defined by the GROUPS factors

DTABLE plots the tables specified by the TABLE parameter (each table displayed in a separate set of plots). The GRAPHICS option controls whether a high-resolution or a line-printer graph is plotted; by default GRAPHICS=high.

The METHOD option controls how each table is plotted in high-resolution graphics, with settings:

points	to plot points at the table values;
linesandpoints	to plot points and join them by lines;
onlylines	to draw lines between the table values;
data	to draw lines between the table values, and then also plot the data values supplied (in a variate) by the DATA parameter;
barchart	to plot the table values as a barchart;
splines	to plot the points together with a smooth spline to show the trend over each group of points; the DFSPLINE specifies the degrees of freedom for the splines; if this is not set, 2 d.f. are used when there are up to 10 points, 3 if there are 11 to 20, and 4 for 21 or more.

By default METHOD=points (and this is the only display available in line-printer graphics).

The XFACTOR parameter defines the factor against whose levels the values of the table are plotted. With a multi-way table, there will be a plot of the table values against the XFACTOR levels for every combination of levels of the other factors classifying the table. The GROUPS parameter specifies factors whose levels are to be included in a single window of the graph. So, for example, if you specify

```
    DTABLE [METHOD=line] Table; XFACTOR=A; GROUPS=B
```

DTABLE will plot the values of Table in a single window with factor A on the x-axis, and a line for each level of the factor B. You can set GROUPS to a pointer to specify several factors to define groups. For example

```
    POINTER [VALUES=B,C] Groupfactors
    DTABLE [METHOD=line] Table; XFACTOR=A; GROUPS=Groupfactors
```

to plot a line for every combination of the levels of factors B and C. Similarly, the TRELLISGROUPS option can specify one or more factors to define a trellis plot. For example,

```
    DTABLE [METHOD=line] Table; XFACTOR=A; GROUPS=B;\
           TRELLISGROUPS=C
```

will produce a plot for each level of C, in a trellis arrangement; each plot will again have factor A on the x-axis, and a line for each level of the factor B. Likewise, the PAGEGROUPS parameter can specify factors whose combinations of levels are to be plotted on different pages. So

```
    DTABLE [METHOD=line] Table; XFACTOR=A; GROUPS=B; PAGEGROUPS=C
```

will again produce a plot for each level of C, but now on separate pages.

If XFACTOR is unset, DTABLE will select the XFACTOR according to the following criteria (in decreasing order of importance): that the factor has no labels, that it has levels that are not the default integers 1 upwards, or that it has more levels than the other factors. If GROUPS is unset, it will be set to all the factors except the XFACTOR. (So, if you want to use either TRELLISGROUPS or PAGEGROUPS, you must also specify XFACTOR and GROUPS.)

The BAR parameter can be set to a scalar to specify an overall (or average) error bar, such as a standard error for differences between any pair of table values. Alternatively, it can be set to a table to specify a different error value, such as an effective standard error, for every table value; DTABLE then plots a bar of the defined size above and below each table value. Finally, it can be set to a pointer containing either two scalars or two tables, specifying the upper and lower positions of the error bar(s). Note, however, that the table setting may be unsuitable for plots other than barcharts when there are GROUPS, as the error bars may overlap each other.

The NEWXLEVELS parameter enables different levels to be supplied for XFACTOR if the existing levels are unsuitable. If XFACTOR has labels, these are used to label the x-axis unless you set option XFREPRESENTATION=levels.

The TITLE, YTITLE and XTITLE parameters can supply titles for the graph, the y-axis and the x-axis, respectively. The symbols, colours and line styles that are used in a high-resolution plot are usually set up by DTABLE automatically. If you want to control these yourself, you should use the PEN directive (6.9.8) to define a pen with your preferred symbol, colour and line style, for each of the groups defined by combinations of the GROUPS factors. The pen numbers should then be supplied to DTABLE, in a variate with a value for each group, using the PENS parameter.

The YTRANSFORM option allows you to include additional axis markings, transformed onto another scale, on the right-hand side of the y-axis. Suppose, for example, the table contains means from an analysis of a variate of percentages that had been transformed to logits. You might then set YTRANSFORM=ilogit (the inverse-logit transformation) to include markings in percentages alongside the logits. The settings are the same as those of the TRANSFORM parameter of AXIS, which is used to add the markings (6.9.7). You can control the colours of the transformed marks and labels, by defining a pen with the required properties, and specifying it with the PENYTRANSFORM option. Otherwise, the default is to plot them in blue.

When there is more than one GROUPS factor, the KEYMETHOD controls whether to use the factor names with their labels (or levels for factors with no labels) or just the labels (or levels) in the key descriptions. The default is to use the names and the labels (or levels). Similarly, the PLOTTITLEMETHOD specifies what to use for the titles of the plots when there is more then one TRELLISGROUPS factor, and the PAGETITLEMETHOD specifies what to use for the titles of the plots when there is more then one PAGEGROUPS factor. You can setKEYMETHOD=* to have no key at all.

The USEAXES option allows you to control various aspects of the axes. First you need to use the XAXIS and YAXIS directives to define them for window 1. Then specify which of the aspects of the axes in window 1 are to be used by DTABLE, by specifying USEAXES with the following settings:

limits	y- and x-axis limits (LOWER and UPPER parameters);
marks	location and labelling of the tick marks (MARKS, LABELS, LDIRECTION, LROTATION, DECIMALS, DREPRESENTATION, and VREPRESENTATION parameters);
mpositions	positions of the tick marks (MPOSITION parameter); and
nsubticks	number of subticks per interval (NSUBTICKS parameter).

By default none are used.

Example 4.11.7 plots the table `Sorted%totals` from Example 4.11.6 with the sorted town factor on the x-axis, and a line for each type of item. Notice that we need to specify new levels `Newx` for the `SortedTowns` to ensure that the towns appear in the sorted order. (The levels of SortedTowns factor are a permutations of the original levels of the `Town` factor, namely 1, 3, 4 and 2. So, if we use these, the towns will appear in the original order.) The graph is shown in Figure 4.11.7.

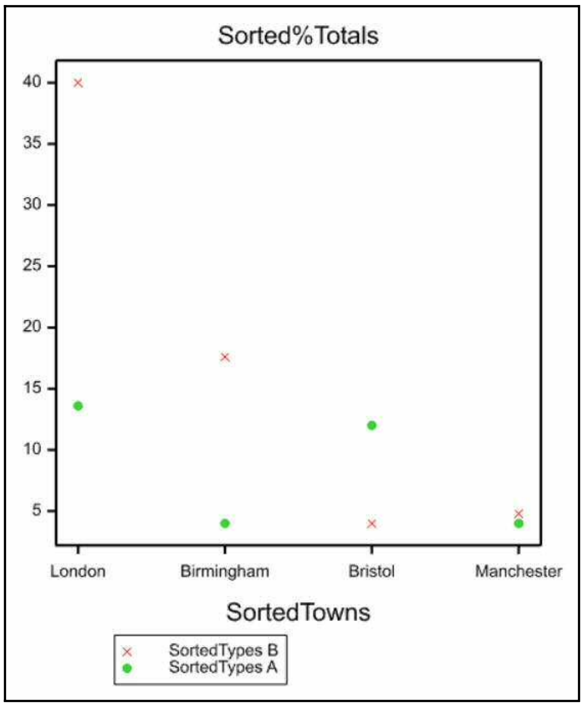

Figure 4.11.7

Example 4.11.7

```
 27  VARIATE [VALUES=1...4] Newx
 28  DTABLE   Sorted%Totals; XFACTOR=SortedTowns; NEWXLEVELS=Newx;\
 29           GROUPS=SortedTypes
```

4.11.8 Interpreting multiple responses: the FMFACTORS procedure

FMFACTORS procedure

Forms a pointer of factors representing a multiple-response (R.W. Payne).

Options

`MRESPONSE` = *pointer*	Pointer with a factor for each code, indicating the units where it occurs in the `CODE` texts or variates
`RESPONSECODES` = *text* or *variate*	Saves the set of distinct multiple-response codes
`CODENULL` = *text* or *variate*	Code(s) used to represent a null value in the `CODE` texts or variates; default `*` or `''`
`EXCLUDENULL` = *string token*	Whether to exclude the factor recording the respondents that made no reply (`yes`, `no`); default `no`
`SUFFIXNULL` = *scalar*	Suffix to use to represent a null value in `MRESPONSE`; default 0
`LABELNULL` = *text*	Label to use to represent a null value in `MRESPONSE`; default `'none'`
`LDIRECTION` = *string token*	How to order the labels from textual codes (`ascending`, `given`); default `asce`

Parameter

`CODE` = *texts*, *variates* or *factors*	Codes from the respondents

Multiple responses occur in surveys as the result of open-ended questions like "Which cities have you visited this year?" or "What languages do you speak?". The easiest way to input these into Genstat is in a set of text vectors. Each text has a unit for every respondent, and the set contains as many texts as the maximum number of the replies from any respondent. Alternatively, if the responses are numerical, they would be input into a set of variates. The `MTABULATE` procedure can form tables with multiple responses. However, these raw codes must first be converted by `FMFACTORS` into a set of factors.

The texts or variates containing the raw data are listed using the `CODE` parameter. You can also supply the raw data in factors. If `CODE` specifies a mixture of texts and factors, `FMFACTORS` uses the labels of the factors (and they must all have labels). Alternatively, if `CODE` specifies a mixture of variates and factors, `FMFACTORS` uses the factor levels. Finally, if `CODE` specifies only factors, `FMFACTORS` will use their labels if they all have labels; otherwise their levels. `FMFACTORS` will give a fault if you specify a mixture of texts and variates.

The multiple-response factors are saved, in a pointer, using the `MRESPONSE` option. The pointer contains a factor for every recorded code, with levels 0 and 1, and corresponding labels `'absent'` and `'present'`. If the codes are textual, the various strings are used as labels of the pointer; while if they are numerical, the numbers are used as the pointer suffixes.

By default, the texts or variates are assumed to contain a missing values for any null response: for example these would occur in the third and fourth text, if there were four `CODE` texts and the respondent concerned had made only two replies. However, you can use the `CODENULL` option to supply alternative codings (for example `'-'` for textual responses).

The `EXCLUDENULL` option controls whether or not the pointer contains a factor to make an explicit record of the respondents that made no replies at all (default `no`). This will be needed if the later tables are to contain a line for "no response". The `SUFFIXLNULL` option specifies the suffix to be used for this factor in the pointer while, for textual codes, the `LABELNULL` option specifies its label in the pointer.

Example 4.11.8 shows the use of `FMFACTORS` to analyse a passenger survey. The participants provide the nationalities, ages, and sexes. They then have five fields in which to record the cities that they have visited recently, and the languages that they speak. A multiple-response pointer is formed in lines 53 and 54 to represent the city responses, and another in lines 55 and 56 to represent the languages. We have set option `EXCLUDENULL=yes` when forming the languages, under the assumption that everyone will be able to speak at least one language! Notice that the nationalities (of which there is only one per participant) can be converted into a factor by using `GROUPS` in the usual way. The first three factors in the multiple-response pointer for languages are printed at the end of the example, together with the original codes, to illustrate how the pointers are formed.

Example 4.11.8

```
   2  " Analysis of a passenger survey."
   3  FACTOR    [LABELS=!t(male,female)] Sex
   4  TEXT      Nationality,Citycode[1...5],Languagecode[1...5]
   5  READ      [PRINT=errors] Nationality,Age,Sex,Citycode[1...5],\
   6            Languagecode[1...5]; FREPRESENTATION=labels
  52  GROUPS    [REDEFINE=yes] Nationality
  53  FMFACTORS [MRESPONSE=Rcity; RESPONSECODES=Cities; CODENULL='-']\
  54            Citycode[]
  55  FMFACTORS [MRESPONSE=Rlanguage; RESPONSECODES=Languages;\
  56            CODENULL='-'; EXCLUDENULL=yes] Languagecode[]
  57  PRINT     [RLWIDTH=20; ORIENTATION=across]\
  58            Rlanguage[1,2,3],Languagecode[]; JUSTIFICATION=left
```

```
Rlanguage['Dutch']   absent     absent     absent     absent     absent
Rlanguage['English'] present    present    present    present    present
Rlanguage['French']  present    present    present    present    present
Languagecode[1]      English    English    English    English    English
Languagecode[2]      French     French     French     French     French
Languagecode[3]      German     German     -          -          -
Languagecode[4]      -          -          -          -          -
Languagecode[5]      -          -          -          -          -

Rlanguage['Dutch']   absent     absent     absent     absent     absent
Rlanguage['English'] present    present    present    present    present
Rlanguage['French']  absent     absent     absent     absent     present
Languagecode[1]      English    English    English    English    English
Languagecode[2]      -          Japanese   -          -          Spanish
Languagecode[3]      -          -          -          -          French
Languagecode[4]      -          -          -          -          -
Languagecode[5]      -          -          -          -          -

Rlanguage['Dutch']   absent     absent     absent     present    present
Rlanguage['English'] present    present    present    present    present
Rlanguage['French']  present    present    present    present    present
Languagecode[1]      English    English    English    Dutch      Dutch
Languagecode[2]      Spanish    French     French     English    English
Languagecode[3]      French     -          -          French     French
Languagecode[4]      -          -          -          -          Italian
Languagecode[5]      -          -          -          -          Spanish

Rlanguage['Dutch']   present    present    present    present    absent
Rlanguage['English'] present    present    present    present    present
Rlanguage['French']  absent     absent     absent     present    present
Languagecode[1]      Dutch      Dutch      Dutch      Dutch      French
Languagecode[2]      English    English    English    English    English
Languagecode[3]      German     -          -          German     -
Languagecode[4]      -          -          -          French     -
Languagecode[5]      -          -          -          -          -

Rlanguage['Dutch']   absent     absent     absent     absent     absent
Rlanguage['English'] present    absent     present    present    present
Rlanguage['French']  present    present    present    present    present
Languagecode[1]      French     French     French     French     French
Languagecode[2]      English    -          English    English    English
Languagecode[3]      -          -          -          -          German
Languagecode[4]      -          -          -          -          -
Languagecode[5]      -          -          -          -          -

Rlanguage['Dutch']   absent     absent     absent     absent     absent
Rlanguage['English'] present    absent     absent     present    absent
Rlanguage['French']  present    present    present    absent     absent
Languagecode[1]      French     French     French     German     German
Languagecode[2]      English    German     German     English    -
Languagecode[3]      -          -          -          -          -
Languagecode[4]      -          -          -          -          -
Languagecode[5]      -          -          -          -          -

Rlanguage['Dutch']   absent     absent     absent     absent     absent
Rlanguage['English'] present    present    present    present    absent
Rlanguage['French']  present    absent     present    absent     absent
Languagecode[1]      German     German     German     German     German
Languagecode[2]      English    English    English    English    -
Languagecode[3]      French     -          French     -          -
Languagecode[4]      -          -          -          -          -
Languagecode[5]      -          -          -          -          -

Rlanguage['Dutch']   absent     absent     absent     absent     absent
Rlanguage['English'] present    present    absent     absent     present
Rlanguage['French']  present    present    present    present    present
Languagecode[1]      French     French     French     German     German
Languagecode[2]      German     German     German     French     French
Languagecode[3]      Italian    English    -          -          English
Languagecode[4]      English    -          -          -          -
Languagecode[5]      -          -          -          -          -
```

4.11.9 Finding multiple responses in free text: the FFREERESPONSEFACTOR procedure

FFREERESPONSEFACTOR procedure

Forms multiple-response factors from free-response data (R.W. Payne).

Options

MRESPONSE = *pointer*	Pointer with a factor for each RESPONSECODE, indicating which of the DATA texts contain that response
RESPONSECODES = *text*	Specifies the codes to look for in the DATA texts
LABELCODES = *text*	Strings to label the factors within the MRESPONSE pointer; default RESPONSECODES
DUPLICATECODES = *factor*	Defines groupings of duplicate or alternative codes within the RESPONSECODES text
EXCLUDENULL = *string token*	Whether to exclude the factor recording which DATA contain none of the RESPONSECODES (yes, no); default no
SUFFIXNULL = *scalars*	Suffix to use to represent the null factor in MRESPONSE; default 0
LABELNULL = *text*	Label to use to represent a the null factor in MRESPONSE; default 'none'
DATAFORMAT = *string token*	Whether the data for the respondents is given line-by-line within the DATA text(s) or whether there is a separate text for each respondent (linebyline, textbytext); default line
CASE = *string token*	Whether to treat the case of letters (small or capital) as significant when searching for the codes (significant, ignored); default igno
MULTISPACES = *string token*	Whether to treat differences between multiple spaces and single spaces as significant, or to treat them all like a single space (significant, ignored); default igno
DISTINCT = *string tokens*	Whether to require each RESPONSECODE to have one or more separators to its left or right within each DATA text (left, right); default left, righ
SEPARATOR = *text*	Characters to use as separators; default ' ,;:.'

Parameter

DATA = *texts*	Information from the respondents

FFREERESPONSEFACTOR supports another way of specifying multiple responses, namely as keywords within free text supplied by each respondent.

The texts from the respondents are specified using the DATA parameter. If option DATAFORMAT=linebyline (the default), there is a single text with a line for each respondent. You can also give a second text, again with a line for each respondent, if you cannot fit all the information for any of the respondents into a single line. Likewise you can specify a third text if you need more than two lines, and so on. Alternatively, if option DATAFORMAT=textbytext, the information from each respondent is contained in a separate text.

The input thus consists of free-form text(s) in which the responses of interest are to be found. For example, in a survey of garden plants, the text might contain the lines

```
 'I grow carrots, cabbages and lettuces. I also have an apple
tree.'
```

Any restrictions on the DATA texts are ignored.

The codes to find within the texts are supplied, in a text, by the `RESPONSECODES` option. If you want to supply alternative codes (for example, synonyms or singular and plural codes), you should put all the alternatives into the `RESPONSECODES` text, and set the `DUPLICATECODES` option to a factor to indicate how the codes are grouped together. For example, `Codes` below contains singular and plural codes for various plants, and `Alternatives` indicates how these belong together

```
TEXT [VALUES=carrot,cabbage,lettuce,potato,tomato,\
   carrots,cabbages,lettuces,potatoes,tomatoes,\
   apple,rose,magnolia,sycamore,'silver birch',\
   apples,roses,magnolias,sycamores,'silver birches'] Codes
FACTOR [LEVELS=10; VALUES=(1...5)2,(6...10)2] Alternatives
```

The pointer of multiple-response factors is saved using the `MRESPONSE` option. By default, the pointer is labelled by the names of the codes (or by the first of each set of codes if there are alternatives). However, you can use the `LABELCODES` option to supply other labels if these are unsuitable (e.g. too long).

The `EXCLUDENULL` option controls whether or not the pointer contains a factor to make an explicit record of the people that gave none of the codes (default `'no'`). This will be needed if the later tables are to contain a line for "no response". The `SUFFIXNULL` option specifies the suffix to be used for this factor in the pointer while, the `LABELNULL` option specifies its label.

`FFREERESPONSEFACTOR` usually ignores the case of letters (small or capital) when looking for the codes. So for example `'Apple'` would be the same as `'apple'`. However, you can set option `CASE=significant` to recognize these differences in case. `FFREERESPONSEFACTOR` usually also treats multiple spaces as the same as a single space, but you can set option `MULTISPACE=significant` to treat these differences as important.

By default, `FFREERESPONSEFACTOR` requires each code to begin either at the start of the `DATA` text or to be preceded in the text by a separator (such as a space or comma). Similarly, it requires each code to end within the text with a separator (or to be at the end of the text). This is requested by the `DISTINCT` option, with its default `DISTINCT=left,right`. However, for example, you can set `DISTINCT=left` if the codes must be separated from other text only to the left (i.e. at the start), or `DISTINCT=*` if they need not be separated at all. The separators are specified by the `SEPARATOR` option.

The two ways of using `FFREERESPONSEFACTOR` are illustrated in Example 4.11.9.

Example 4.11.9

```
   2  TEXT    [VALUE='In my garden I grow carrots, cabbages and lettuces.',\
   3          'Vegetables: potatoes, carrots.',\
   4          'I just have some concrete where I park the car.']\
   5          Data
   6  TEXT    [VALUES=carrot,cabbage,lettuce,potato,tomato,\
   7          carrots,cabbages,lettuces,potatoes,tomatoes,\
   8          apple,rose,magnolia,sycamore,'silver birch',\
   9          apples,roses,magnolias,sycamores,'silver birches'] Codes
  10  FACTOR  [LEVELS=10; VALUES=(1...5)2,(6...10)2] Alternatives
  11  FFREERESPONSEFACTOR [MRESPONSE=Vegetables; RESPONSECODES=Codes;\
  12          DUPLICATECODES=Alternatives] Data
  13  PRINT   Vegetables[]

Vegetables['none'] Vegetables['carrot'] Vegetables['cabbage']
         responded             present              present
         responded             present               absent
       no response              absent               absent

 Vegetables['lettuce'] Vegetables['potato'] Vegetables['tomato']
               present               absent               absent
                absent              present               absent
                absent               absent               absent
 Vegetables['apple'] Vegetables['rose'] Vegetables['magnolia']
              absent             absent                 absent
              absent             absent                 absent
              absent             absent                 absent

 Vegetables['sycamore'] Vegetables['silver birch']
                 absent                     absent
                 absent                     absent
                 absent                     absent

  14  TEXT    [VALUE='In my garden I grow carrots, cabbages and lettuces.',\
  15          'I also have an apple tree, a rose bush and a magnolia tree.']\
  16          Garden[1]
  17  &       [VALUE='Vegetables: potatoes, carrots.',\
  18          'Trees: sycamore, silver birch.'] Garden[2]
  19  &       [VALUE='I just have some concrete where I park the car.']\
  20          Garden[3]
  21  FFREERESPONSEFACTOR [MRESPONSE=Plant; RESPONSECODES=Codes;\
  22          DUPLICATECODES=Alternatives; DATAFORMAT=textbytext] Garden[]
  23  PRINT   Plant[]; FIELD=18

     Plant['none']   Plant['carrot']  Plant['cabbage']  Plant['lettuce']
         responded           present           present           present
         responded           present            absent            absent
       no response            absent            absent            absent

   Plant['potato']   Plant['tomato']    Plant['apple']     Plant['rose']
            absent            absent           present           present
           present            absent            absent            absent
            absent            absent            absent            absent

 Plant['magnolia'] Plant['sycamore'] Plant['silver birch']
           present            absent            absent
            absent           present           present
            absent            absent            absent
```

4.11.10 Tabulation with multiple responses: the **MTABULATE** procedure

MTABULATE procedure

Forms tables classified by multiple-response factors (R.W. Payne).

Options

PRINT = *string token*	Controls printed output (counts, totals, nobservations, means, minima, maxima, variances, quantiles, sds, skewness, kurtosis, semeans, seskewness, sekurtosis); default * i.e. none
CLASSIFICATION = *factors*	Non multiple-response factors classifying the tables
MRESPONSE = *pointers*	Pointers to factors defining the multiple-responses for the tables
MRFACTOR = *identifiers*	Identifier of factors to index the sets of multiple responses in the tables
COUNTS = *table*	Saves a table counting the number of units with each factor combination; default *
MARGINS = *string token*	Whether the tables should be given margins (yes, no); default no
WEIGHTS = *variate*	Weights to be used in the tabulations; default * indicates that all units have weight 1
PERCENTQUANTILES = *scalar* or *variate*	Percentages for which quantiles are required; default 50

	i.e. median

Parameters

DATA = *variates*	Data values to be tabulated
TOTALS = *tables*	Tables to contain totals
NOBSERVATIONS = *tables*	Tables containing the numbers of non-missing values in each cell
MEANS = *tables*	Tables of means
MINIMA = *tables*	Tables of minimum values in each cell
MAXIMA = *tables*	Tables of maximum values in each cell
VARIANCES = *tables*	Tables of cell variances
QUANTILES = *tables* or *pointers*	Table to contain quantiles at a single PERCENTQUANTILE, or pointer of pointers to tables for several PERCENTQUANTILES
SDS = *tables*	Tables of standard deviations
SKEWNESS = *tables*	Tables of skewness coefficients
KURTOSIS = *tables*	Tables of kurtosis coefficients
SEMEANS = *tables*	Tables of standard errors of means
SESKEWNESS = *tables*	Tables of skewness coefficients
SEKURTOSIS = *tables*	Tables of kurtosis coefficients

Once the multiple responses in a survey have been processed by FMFACTORS, to form each set into a pointer containing a factor for each possible response code, the results can be tabulated using the MTABULATE procedure.

The multiple responses for the tables are specified by the MRESPONSE option, while any ordinary factors are specified by the CLASSIFICATION option. The MARGINS option indicates whether or not the tables are to contain margins. For the multiple responses, these represent summaries not over the responses but over the respondents (who may each have given several responses). MTABULATE needs to generate an ordinary factor to classify the dimension of the tables corresponding to each set of multiple responses. You can supply identifiers for these factors (thus allowing them to be accessed outside the procedure), using the MRFACTOR option.

The other options and parameters are similar to those of the TABULATE directive. The COUNTS option can save a table containing the frequencies of the various responses. The DATA parameter provides information about the respondents who made the multiple responses. (So, for example, you could set DATA to the incomes of the respondents and then tabulate the average incomes of the people who have visited each of the cities.) The other parameters allow you to save the various types of numerical summary: totals, numbers of non-missing values, means, minima, maxima, variances, quantiles, standard deviations, skewness and kurtosis coefficients and (within-cell) standard errors of means, skewness and kurtosis.

The PERCENTQUANTILES option specifies which quantiles you want. By default just the median (the 50% quantile) is produced. However, you can set PERCENTQUANTILES to a scalar to request another percentage point, or to a variate to request several. The QUANTILE parameter will then return a pointer with length equal to the required number of quantiles, instead of a single table.

The PRINT option allows you to print the tables (as well as, or instead of, saving them). By default nothing is printed.

Example 4.11.10 continues the analysis of the survey in Example 4.11.8. Notice that a default identifier _['Rcity'] factor is used to classify the multiple-response dimension of the first table; this will exist only within the procedure. For the second table the identifier Language is specified (line 61), and this will be a factor in the main program.

Example 4.11.10

```
 59  MTABULATE [PRINT=Counts; CLASSIFICATION=Nationality; MRESPONSE=Rcity]

                   Counts

 Nationality      British       Dutch      French      German        Swiss
  _['Rcity']
        none            0           0           0           1            0
   Amsterdam            1           0           1           1            0
      Athens            1           0           0           0            0
   Barcelona            5           0           0           0            0
      Berlin            1           0           2           0            0
    Brussels            1           2           0           2            1
    Capetown            2           0           0           0            0
  Copenhagen            2           0           2           0            0
      Dublin            2           0           0           0            0
   Edinburgh            0           0           2           0            1
    Florence            0           2           0           0            3
   Frankfurt            0           1           0           1            0
      Geneva            1           0           0           0            0
    Helsinki            1           0           0           0            0
      Lisbon            2           0           0           0            0
      London            0           3           4           4            1
  Luxembourg            0           1           0           1            1
      Madrid            2           2           0           1            0
      Oxford            0           0           2           0            0
       Paris            5           1           0           2            1
        Pisa            0           0           0           0            1
        Rome            0           2           0           0            3
     Seville            0           0           0           0            1
      Venice            0           0           0           0            2

 60  MTABULATE [PRINT=Mean; CLASSIFICATION=Sex,Nationality;\
 61            MRESPONSE=Rlanguage; MRFACTOR=Language] Age

                                Means

            Nationality   British     Dutch    French    German     Swiss
  Language          Sex
     Dutch         male         *     33.75         *         *         *
                 female         *     40.50         *         *         *
   English         male     34.14     33.75     32.50     41.67     41.00
                 female     34.00     40.50     32.25     30.50     29.00
    French         male     33.40     28.33     39.25     45.00     38.00
                 female     33.75         *     34.40     24.00     35.00
    German         male     39.00     26.00     35.00     41.67     38.00
                 female         *     33.00     43.00     49.75     35.00
   Italian         male         *     34.00         *         *     45.00
                 female         *         *         *         *         *
  Japanese         male         *         *         *         *         *
                 female     31.00         *         *         *         *
   Spanish         male     51.00     34.00         *         *         *
                 female     50.00         *         *         *         *
```

4.12 Operations on trees

Tree structures are used to represent hierarchical structures like classification trees, identification keys and regression trees. These types of tree can be constructed by special-purpose procedures BCLASSIFICATION (2:6.20.1), BKEY (2:6.21.1) and BREGRESSION (2:3.9.1), respectively. Most people will use only these special-purpose procedures, and their associated procedures (for example BRCONSTRUCT, BRDISPLAY and BRPREDICT for regression trees). They will not need to operate on trees directly, nor to be aware of how they are formed, stored or manipulated.

The procedures, however, are based on a suite of directives, functions and procedures described in this section. These provide the tool kit not only for the officially-supported tree facilities but also for user enhancements and extensions. Initially, we describe the utility procedures BPRINT and BGRAPH for displaying trees (used by BCDISPLAY, BRDISPLAY and so on). We then cover the various utility commands for constructing and modifying trees.

The tree structure is like a real tree, which starts from a root and then splits into branches, except that it is usually viewed as growing downwards instead of upwards. The branch-points in the tree are known as *nodes*, with the initial node being called the *root* (as in a real tree). There is also a node at the end of each branch, known as its *terminal node*. In Genstat a tree is similar to a pointer, with an element for each node. These elements are the identifiers of data structures which can be used to store information about the nodes. Usually the data structures will be pointers, so that several pieces of information can be stored for each node, but the precise contents depend on the type of tree.

Each node thus has a number, corresponding to the index of its element in the tree structure. The root is always numbered one, and this is the only node that the tree contains when it is declared by TREE (2.8). Further nodes can be added by the directives BGROW (4.12.3) and BJOIN (4.12.5), which form branches from a terminal node or join another tree to a terminal node, respectively. The converse process of cutting a tree at a defined node and discarding the nodes and information below it is provided by the BCUT directive (4.12.4), which can also duplicate a tree.

The numbering of the nodes depends mainly on the order in which they have been added to the tree (although BCUT does allow you to renumber them into a "standard" order). The idea is that you obtain the numbers of the nodes below the root (node number one) by using the various tree functions described in Section 4.2.11. These are illustrated in Example 4.12.5.

4.12.1 Printing a tree: the BPRINT procedure

BPRINT procedure

Displays a tree (R.W. Payne).

Option

PRINT = *string tokens*	Controls printed output (indented, bracketed, labelleddiagram, numbereddiagram); default inde

Parameter

TREE = *trees*	Trees to be displayed

BPRINT can print a tree in various formats. The tree is specified by the TREE parameter, and the PRINT option indicates what output is required, with settings:

bracketed	display as used to represent an identification key in "bracketed" form (printed node by node);
indented	display as used to represent an identification key in "indented" form (printed branch by branch);
labelleddiagram	diagrammatic display including the node labels;
numbereddiagram	diagrammatic display with the nodes labelled by their numbers.

The use of BPRINT is illustrated in the Examples 4.12.3, 4.12.4 and 4.12.5.

4.12.2 Plotting a tree: the **BGRAPH** procedure

BGRAPH procedure

Plots a tree (R.W. Payne).

Option

`SIZE` = *scalar*	Provides a multiplier by which to scale the node labels

Parameters

`TREE` = *trees*	Trees to be plotted
`XTERMINAL` = *scalars* or *variates*	X-spacing (scalar) or x-values (variate) for the terminal nodes of each tree; default 2

The tree to be plotted is specified by the `TREE` parameter. `BGRAPH` arranges the nodes with the root at the top and the terminal nodes at the bottom of the plot. The terminal nodes are arranged automatically across the screen, but the x-coordinates can be specified explicitly using the `XTERMINAL` parameter. The `SIZE` option allows the size of the node labels to be adjusted by a scaling factor (default 1).

4.12.3 Extending a tree: the **BGROW** directive

BGROW directive

Adds new branches to a node of a tree.

No options

Parameters

`TREE` = *trees*	Trees to be extended
`NODE` = *scalars*	Node at which to extend each tree
`NBRANCHES` = *scalars*	Number of branches to add to each node; default 2
`POSITION` = *scalars*	Position at which to add the branches to each node; default `*` i.e. after all the current braches from the node
`NEWNODES` = *variates*	Returns the number(s) allocated to the new nodes

`BGROW` provides the basic tree utility of adding new branches at a node, which is used for example by the `BCONSTRUCT` procedure (4.12.6). The tree to be extended is specified by the `TREE` parameter, and the `NODE` parameter indicates the node at which the new branches are to be added. The `NBRANCHES` parameter specifies the number of branches to add. The `POSITION` specifies where to add them if the node is a non-terminal node; by default they are added after all the branches currently from the node. The `NEWNODES` parameter saves a variate containing the numbers of the new nodes (i.e. the terminal nodes at the ends of the new branches).

The use of `BGROW` is shown in Example 4.12.3. Notice that the tree element at each node is set up as a pointer with one element labelled as `'label'`. This is then used by `BPRINT` to label the tree in labelled-diagram format.

Example 4.12.3

```
" Declare the original tree."
TREE      T
" Define texts to use as labels for the nodes."
TEXT      Lab[1...26]; VALUES=\
          'a','b','c','d','e','f','g','h','i','j','k','l','m',\
   7           'n','o','p','q','r','s','t','u','v','w','x','y','z'
   8  " Define information at root to be a pointer
  -9    with a single element called 'label'."
  10  POINTER   [NVALUES=!t(label)] T[1]
  11  " Set that element to the first value of Lab, i.e. 'a'."
  12  TEXT      T[1]['label']; VALUE=Lab[1]
  13  " Display the tree - first with labels of nodes, then with numbers."
  14  BPRINT    [PRINT=labelleddiagram,numbereddiagram] T

Tree with labels

a


Tree diagram

1

  15  " Extend the tree by forming 3 branches from node 1 (root)."
  16  BGROW     T; NODE=1; NBRANCH=3; NEWNODES=Gnew
  17  " Define the information for the new nodes."
  18  POINTER   [NVALUES=!t(label)] T[#Gnew]
  19  TEXT      T[#Gnew]['label']; VALUE=Lab[#Gnew]
  20  " Display the extended tree."
  21  BPRINT    [PRINT=labelleddiagram,numbereddiagram] T

Tree with labels

a  b
-> c
-> d


 Tree diagram

1  2
-> 3
-> 4

  22  " Find the node number of the first terminal node "
  23  CALCULATE N1 = BTERMINAL(T; 0)
  24  " and then the second terminal node."
  25  CALCULATE N2 = BTERMINAL(T; N1)
  26  PRINT     N1,N2; DECIMALS=0

          N1          N2
           2           3

  27  " Extend the tree by adding 2 branches at the second
 -28    and then the first terminal node."
  29  BGROW     T; NODE=N2; NBRANCH=2; NEWNODES=Gnew2
  30  " Define the information for the new nodes."
  31  POINTER   [NVALUES=!t(label)] T[#Gnew2]
  32  TEXT      T[#Gnew2]['label']; VALUE=Lab[#Gnew2]
  33  BGROW     T; NODE=N1; NBRANCH=2; NEWNODES=Gnew1
  34  POINTER   [NVALUES=!t(label)] T[#Gnew1]
  35  TEXT      T[#Gnew1]['label']; VALUE=Lab[#Gnew1]
  36  " Display the extended tree."
  37  BPRINT    [PRINT=labelleddiagram,numbereddiagram] T

Tree with labels

a  b  g
   -> h
-> c  e
   -> f
-> d


 Tree diagram

1  2  7
   -> 8
-> 3  5
   -> 6
-> 4
```

4.12.4 Removing branches from a tree: the BCUT directive

BCUT directive

Cuts a tree at a defined node, discarding the nodes and information below it.

Option

RENUMBER = *string token*	Whether or not to renumber the nodes of the tree (yes, no); default no

Parameters

TREE = *trees*	Trees to be cut
NODE = *scalars*	Node at which to cut each tree
NEWTREE = *trees*	New trees with the information cut; if unspecified, the new tree replaces the original tree
CUTTREE = *trees*	Tree formed from the branches cut from the original tree
OLDNODES = *variates*	Mapping from old nodes to node numbers in a renumbered new tree (as positive numbers) or to nodes in the CUTTREE (as negative numbers)
NEWNODES = *variates*	Mapping from new node numbers in a renumbered tree to the original nodes
CUTNODES = *variates*	Mapping from node numbers in the CUTTREE tree to the original nodes

BCUT provides the basic tree utility of removing an unwanted branch, which is used for example by the BPRUNE procedure. The tree to be cut is specified by the TREE parameter, and the NODE parameter indicates the node at which the cut is to be made. The NEWTREE parameter can supply the identifier for the new tree (after removing all the nodes below NODE); if this is not specified, the new tree replaces the original tree. The subtree below NODE can also be saved (as a tree in its own right, with NODE as the root) using the CUTTREE parameter.

The OLDNODES parameter can save a variate containing a mapping from the old nodes to the new nodes. If the node is a member of the new tree the variate contains the number of that node in the NEWTREE, while if it is one of the nodes that are deleted the variate contains -1 multiplied by its number in the CUTTREE. As far as OLDNODES is concerned NODE is regarded as a member of the NEWTREE.

The NEWNODES parameter can save a variate containing the converse mapping from the NEWTREE to the original tree. There is an element for each new node, containing the number of the equivalent node in the original tree. Similarly, the CUTNODES parameter can save a mapping from the CUTTREE to the original tree.

Example 4.12.4 continues Example 4.12.3. BCUT removes one of the sub-branches of tree T. Notice that the root of the cut-tree T2 initially contains the same information as at the node N2 where the tree was cut. This is then replaced in lines 48-50. Also notice the use of the RENUMBER option in line 54 to renumber the nodes of the new tree T3.

Example 4.12.4

```
  38  " Remove the branches below N2, saving these as tree T2;
 -39    also save and print the node mapping variates."
  40  BCUT      T; NODE=N2; CUTTREE=T2; OLDNODES=Oldn;\
  41           NEWNODES=Newn; CUTNODES=Cutn
  42  PRINT     [ORIENT=across] Oldn,Newn,Cutn; FIELD=3; DECIMALS=0

        Oldn  1  2  5  6 -2 -3  3  4

        Newn  1  2  7  8  3  4

        Cutn  3  5  6

  43  " Display the modified tree, and the cut-tree."
  44  BPRINT    [PRINT=labelleddiagram,numbereddiagram] T,T2

Tree with labels

a  b  g
   -> h
-> c
-> d


Tree diagram

1  2  7
   -> 8
-> 3
-> 4


Tree with labels

c  e
-> f


Tree diagram

1  2
-> 3

  45  " Redefine the root of the cut tree so that it no
 -46    longer shares the same information pointer as the
 -47    node where the cut was made in the original tree."
  48  POINTER   [NVALUES=!t(label)] T2root
  49  TEXT      T2root['label']; VALUE='t2root'
  50  ASSIGN    T2root; T2; 1
  51  BPRINT    [PRINT=labelleddiagram] T2

Tree with labels

t2root e
->     f

  52  " Use BCUT to form T3 as a duplicate of T
 -53    but with renumbered nodes."
  54  BCUT      [RENUMBER=yes] T; NEWTREE=T3
  55  BPRINT    [PRINT=labelleddiagram,numbereddiagram] T3

Tree with labels

a  b  g
   -> h
-> c
-> d


Tree diagram

1  2  3
   -> 4
-> 5
-> 6
```

4.12.5 Joining a tree onto another: the BJOIN directive

BJOIN directive

Extends a tree by joining another tree to a terminal node.

No options

Parameters

TREE = *trees*	Trees to be extended
NODE = *scalars*	Node at which to join the tree
JOINTREE = *trees*	Trees to be joined onto the tree
NEWNODES = *variates*	New node numbers allocated to each node in JOINTREE in the new tree

BJOIN provides the basic tree utility of joining a tree to the terminal node of a tree. The tree to be extended is specified by the TREE parameter, and the NODE parameter indicates the node at which the tree is to be joined. The JOINTREE parameter specifies the tree to be joined onto the tree, and the NEWNODES parameter saves a variate containing the numbers of the nodes of the JOINTREE in the new tree.

Example 4.12.5 first joins the tree T2, cut from tree T in Example 4.12.4, onto tree T3 at node number 4. It also illustrates some of the tree functions.

Example 4.12.5

```
  56  " Join tree T2 onto node 4 of T3; save and print the
 -57    numbers of the joined nodes in the revised tree."
  58  BJOIN     T3; NODE=4; JOINTREE=T2; NEWNODES=Jnew
  59  BPRINT    [PRINT=labelleddiagram,numbereddiagram] T3

Tree with labels

a  b  g
   -> h  e
      -> f
-> c
-> d


Tree diagram

1  2  3
   -> 4  7
      -> 8
-> 5
-> 6

  60  PRINT     Jnew; DECIMALS=0

        Jnew
           4
           7
           8

  61  " Tree functions: all nodes below node 2,"
  62  CALCULATE Below  = BBELOW(T3; 0; 0)
  63  PRINT     Below; DECIMALS=0

       Below
           1
           2
           5
           6
```

```
          3
          4
          7
          8

 64  " all terminal nodes below node 2,"
 65  CALCULATE Below0 = BBELOW(T3; 0; 0)
 66  PRINT     Below0; DECIMALS=0

     Below0
          1
          2
          5
          6
          3
          4
          7
          8

 67  " first three terminal nodes,"
 68  CALCULATE N1 = BTERMINAL(T3; 0)
 69  &         N2 = BTERMINAL(T3; N1)
 70  &         N3 = BTERMINAL(T3; N2)
 71  PRINT     N1,N2,N3; DECIMALS=0

         N1          N2          N3
          3           7           8

 72  " nodes and branches on path to N3,"
 73  CALCULATE Pn3 = BPATH(T3; N3)
 74  &         Ln3 = BBRANCHES(T3; N3)
 75  PRINT     Pn3,Ln3; DECIMALS=0

        Pn3         Ln3
          1           1
          2           2
          4           2
          8           0

 76  " depth and number of branches at node 2,"
 77  CALCULATE Nn2 = BNBRANCHES(T3; 2)
 78  &         Dn2 = BDEPTH(T3; 2)
 79  PRINT     Nn2,Dn2; DECIMALS=0

        Nn2         Dn2
          2           2

 80  " next nodes on branches 1-3 from node 1,
-81    and branch 1 from node 2."
 82  PRINT     BNEXT(T3; 1; 1); DECIMALS=0

BNEXT(((T3; 1); 1))
          2

 83  PRINT     BNEXT(T3; 1; 2); DECIMALS=0

BNEXT(((T3; 1); 2))
          5

 84  PRINT     BNEXT(T3; 1; 3); DECIMALS=0

BNEXT(((T3; 1); 3))
          6

 85  PRINT     BNEXT(T3; 2; 1); DECIMALS=0

BNEXT(((T3; 2); 1))
          3

 86  " Scan the tree, taking the nodes in standard order."
 87  SCALAR    Scan[0]; value=0
 88  CALCULATE Scan[1] = BSCAN(T3; Scan[0])
 89  &         Scan[2] = BSCAN(T3; Scan[1])
```

```
&         Scan[3] = BSCAN(T3; Scan[2])
&         Scan[4] = BSCAN(T3; Scan[3])
&         Scan[5] = BSCAN(T3; Scan[4])
&         Scan[6] = BSCAN(T3; Scan[5])
&         Scan[7] = BSCAN(T3; Scan[6])
&         Scan[8] = BSCAN(T3; Scan[7])
&         Scan[9] = BSCAN(T3; Scan[8])
PRINT     Scan[1...9]; FIELD=8; DECIMALS=0

 Scan[1] Scan[2] Scan[3] Scan[4] Scan[5] Scan[6] Scan[7] Scan[8] Scan[9]
       1       2       3       4       7       8       5       6       *
```

4.12.6 Constructing a tree: the BCONSTRUCT procedure

BCONSTRUCT procedure

Constructs a tree (R.W. Payne).

Option

PRINT = *string token*	Whether to print monitoring information (monitoring); default * i.e. none

Parameters

TREE = *trees*	Saves the trees that have been constructed
DATA = *identifiers*	Data available for constructing the trees

BCONSTRUCT is a utility procedure that is used by the tree procedures like BCLASSIFICATION, BKEY and BREGRESSION to construct trees. The DATA parameter of BCONSTRUCT supplies a pointer containing the information required to construct the tree. The TREE parameter saves the tree that has been constructed, and the PRINT option can be set to monitoring to produce monitoring information during construction.

BCONSTRUCT calls a procedure BSELECT to determine the test to be performed at each node of the tree. Customized versions of this procedure are available for each type of tree, and are accessed automatically along with the top-level procedure for the type of tree concerned. BCONSTRUCT is thus completely general - and can be used for other types of tree simply by providing an appropriate version of BSELECT. Within BSELECT you can use any of the Genstat commands, such as CALCULATE or TABULATE, to decide which test to use. Also, an efficient implementation of the standard selection criteria for regression and classification trees is provided by directive BASSESS (4.12.7).

BSELECT has no options. Its parameters are as follows.

DATA = *pointer*	Data for constructing the tree (as provided by the DATA parameter of BCONSTRUCT)
TESTS = *pointer*	Tests already made between the root and the current node
BRANCHES = *variate*	Branches taken at each previous node
LABEL = *text*	Returns a label to put onto the node
NEWTEST = *scalar* or *expression*	New test to be done at the node (expression), or identification made at the node (scalar) if no new test selected
NBRANCH = *scalar*	Returns the number of branches to insert below the node
ADDITIONAL = *pointer*	Other information to store at the node
LADDITIONAL = *text*	Labels for the other information

After BSELECT has selected a test, the tree is extended by the BGROW directive, function BTERMINAL is used to find the next terminal node, and functions BPATH and BBRANCHES are used to ascertain the nodes and branches between that node and the root.

4.12.7 Assessing potential splits: the BASSESS directive

BASSESS directive

Assesses potential splits for regression and classification trees.

Options

Y = *variate* or *factor*	Response variate for a regression tree, or factor specifying the groupings for a classification tree
SELECTED = *dummy*	Returns the identifier of X variate or factor used in the best split
TESTSPLIT = *expression structure*	Logical expression representing the best split
MAXSPLITPOINT = *scalar* or *variate*	When SELECTED is a variate or a factor with ordered levels this returns a scalar containing the boundary between the two splits, when the SELECTED is a factor with unordered levels it returns a variate containing the levels allocated to the first split
MAXCRITERION = *scalar*	Maximum value obtained for the selection criterion
NOSELECTION = *scalar*	Returns the value 1 if no split has been selected, otherwise 0
FMETHOD = *string token*	Selection method to use when Y is a factor (Gini, MPI); default Gini
ANTIENDCUTFACTOR = *string token*	Anti-end-cut factor to use when Y is a factor (classnumber, reciprocalentropy); default * i.e. none
WEIGHTS = *variate*	Weights; default * i.e. all weights 1
TOLERANCE = *scalar*	Tolerance multiplier used e.g. to check for equality of x-values; default * i.e. set automatically for the implementation concerned

Parameters

X = *variates* or *factors*	Variables available to make the split
ORDERED = *string tokens*	Whether factor levels are ordered (yes, no); default no
SPLITPOINT = *scalars* or *variates*	Saves details of the best split found for each X variable; when X is a variate or a factor with ordered levels this returns a scalar containing the boundary between the two splits, when the X is a factor with unordered levels it returns a variate containing the levels allocated to the first split
CRITERIONVALUE = *scalars*	Saves the value of the selection criterion for the best split found for each X variable

BASSESS selects splits for use when constructing classification or regression trees. The Y option specifies the factor defining the groupings for a classification tree, or the response variate for a regression tree. The x-variables that are available to make the split are supplied by the X parameter. They can be variates, or factors with either ordered or unordered levels as indicated by the ORDERED parameter. For example, a factor called Dose with levels for example 1, 1.5, 2 and 2.5 would usually be treated as having ordered levels, whereas levels labelled 'Morphine', 'Amidone', 'Phenadoxone' and 'Pethidine' of a factor called Drug would be regarded as unordered.

In a regression tree, the accuracy of each node is the squared distance of the values of the response variate from their mean for the observations at the node, divided by the total number of observations. The potential splits are assessed by their effect on the accuracy, that is the difference between the initial accuracy and the sum of the accuracies of the two successor nodes resulting from the split.

For a classification tree, the FMETHOD option allows one of two selection criteria to be requested. The default setting, Gini, uses the change in Gini information:

$$G = (1 - \sum_k \alpha_k^2) - (\sum_k \beta_{1k}) \times (1 - \sum_k \beta_{1k}^2) - (\sum_k \beta_{2k}) \times (1 - \sum_k \beta_{2k}^2)$$

where α_k is the proportion of individuals in the original set that are in group *k*, and β_{ik} is the proportion of individuals in successor set *i* (*i* = 1 or 2) that are in group *k*. The aim here is to split the individuals into sets to maximize differences between the within-set group probabilities. An equivalent formula (Taylor & Silverman 1993, Section 4) is

$$G = (p_1 \times p_2) \times \{ \sum_k \beta_{1k}^2 + \sum_k \beta_{2k}^2 - \sum_k (\beta_{1k} \times \beta_{2k}) \}$$

where $p_i = \sum_k \beta_{ik}$. The alternative MPI (*mean posterior improvement*) criterion concentrates more on making the group probabilities differ between the successor sets:

$$MPI = (p_1 \times p_2) \times \{ 1 - \sum_k ((\beta_{1k} \times \beta_{2k}) / (\beta_{1k} + \beta_{2k})) \}$$

Taylor & Silverman (1993) note that the term $(p_1 \times p_2)$ aims to generate successor sets of similar size, and refer to it as the *anti-end-cut factor* because it aims to avoid sets being produced with only a small number of individuals. The ANTIENDCUTFACTOR option allows you to request use of an *adaptive* anti-end-cut factor as devised by Taylor & Silverman (1993, Section 5). This has the form

$$\min \{ p_1 \times p_2, p_{low} \times (1 - p_{low}) \}$$

where p_{low} is the reciprocal of the number of groups in the initial set for the classnumber setting of the ANTIENDCUTFACTOR option, and

$$\min \{ 0.5, 1 / (\sum_k \alpha_k^2) \}$$

for the reciprocalentropy setting. The idea is to encourage splits that lead to terminal modes – and to take accounts of the fact that these are more likely to be generated as the number of groups becomes small.

The SPLITPOINT parameter can be used to save details of the best split found for each X variable. When X is a variate or a factor with ordered levels, this returns a scalar containing the boundary between the two splits. Alternatively, when X is a factor with unordered levels, it returns a variate containing the levels allocated to the first split. The CRITERIONVALUE parameter saves the value of the selection criterion for the best split found for each X variable.

The SELECTED option can be set to a dummy to store the identifier of the X variate or factor used in the best split, and the MAXSPLITPOINT option can save details of the best split, similarly to the SPLITPOINT parameter. The MAXCRITERION option saves the maximum value obtained for the selection criterion, and the NOSELECTION saves a scalar containing the value 0 if a split could be selected or 1 if no further splitting was possible. You can save a logical expression representing the best split using the TESTSPLIT option. So, for example, you can put

```
BASSESS [Y=Yvar; TESTSPLIT=Test; ...]
RESTRICT Yvar; #Test == 1
PRINT Yvar
```

to print the y-values of the individuals in the first successor set. BASSESS takes account of restrictions on Y or on any of the X variates or factors. So you also could now use BASSESS to find the best split on that set.

The WEIGHTS option can supply a variate of weights for the observations. This could be used to supply prior probabilities, or to emphasize units that are perceived as being especially important.

Finally, the TOLERANCE option can be used to modify the tolerance multiplier used internally for example to check for equality of x-values. By default this is set automatically to a value appropriate for the Genstat implementation concerned.

4.12.8 Pruning a tree: the BPRUNE procedure

BPRUNE procedure

Prunes a tree using minimal cost complexity (R.W. Payne).

Option

PRINT = *string tokens*	Controls printed output (graph, table, monitoring); default tabl

Parameters

TREE = *trees*	Trees to be pruned
ACCURACY = *pointers*	Accuracy values for the nodes of each tree; default is to use those stored with the tree
NEWTREES = *pointers*	Saves the trees generated during the pruning of each tree
RTPRUNED = *variates*	Accuracy of the pruned trees of each tree
NTERMINAL = *variates*	Number of terminal nodes in the pruned trees of each tree

The construction of a classification tree or a regression tree generally results in *over fitting*, that is it continues to extend the branches of the tree beyond the point that can be justified statistically. The solution is to prune the tree to remove the uninformative sub-branches.

The tree to be pruned is specified by the TREE parameter. BPRUNE assumes that there is an *accuracy* figure R(t) available for each node t of the tree. By default this is assumed to be stored with the tree itself, but you can specify other values using the ACCURACY parameter. This should be set to a pointer whose suffixes are the same as the numbers of the nodes in the tree, and whose elements are scalars storing the relevant accuracy values.

For a classification tree the accuracy measures the impurity of the subset of individuals at that node (how far it is from being homogeneous i.e. with individuals from a single group). For a regression tree it is the average squared distance of the values of the response variate from their mean for the subset of observations at that node. The accuracy R(T) of the whole tree T is the sum of the accuracies of its terminal nodes.

BPRUNE uses the principle of minimal cost complexity (Breiman *et al.* 1984, Chapter 3) to produce a sequence of pruned trees. At each stage it prunes at the node which is the *weakest link*. Define R(T_t) to be the accuracy of the subtree with root at node t, and nterm(t) to be its number of terminal nodes. The weakest link is then the node for which

$$(\mathrm{R}(t) - \mathrm{R}(T_t)) / (\mathrm{nterm}(t) - 1)$$

is a minimum. The pruned trees can be saved, in a pointer, using the NEWTREES parameter. Their accuracies can be saved (in a variate) using the RTPRUNED parameter, and their numbers of terminal nodes can be saved (also in a variate) using the NTERMINAL parameter.

Printed output is controlled by the PRINT option, with settings:

graph	plots RTPRUNED against NTERMINAL;
table	prints a table with RTPRUNED and NTERMINAL;
monitoring	provides monitoring information during the pruning.

The plot of RTPRUNED against NTERMINAL demonstrates the trade-off between accuracy and complexity (number of terminal nodes). It should show an initial rapid decrease, followed by a long flat region, and then often a gradual increase. The aim is to select a tree that is accurate but not over-complex. One possibility is to take the tree at the point where the graph levels off. However, RTPRUNED contains only an estimate of the accuracy of the trees. So Breiman *et al.* (1984) recommend taking a tree a little above that (in fact at one standard error of RTPRUNED above the minimum point in the graph: see Chapters 3 and 11). In practice though a small amount of over-fitting should not be a problem, so the exact choice of pruned tree should not be crucial.

The use of BPRUNE is illustrated in 2:3.9.3 and 2:6:18.3.

4.12.9 Identification using a tree: the **BIDENTIFY** directive

BIDENTIFY directive

Identifies specimens using a tree.

Options

TREE = *tree*	Specifies the tree
TESTELEMENT = *scalar*	Specifies which element of the pointer of information stored at each node of the tree contains the test to be done there to determine which subsequent branch to take
TERMINALNODES = *scalar*, *variate* or *pointer*	Scalar or variate saving the number or numbers of the terminal nodes reached by a single specimen, or pointer of scalars or variates saving the numbers of the terminal nodes reached by several specimens

Parameters

X = *factors* or *variates*	Variables involved in the tests performed in the tree
VALUES = *scalars*, *variates* or *texts*	Values of the variables for the specimens to be identified

BIDENTIFY identifies specimens using a classification tree, or a regression tree, or an identification key as constructed by procedures BCLASSIFICATION (2:6.21.1), BREGRESSION (2:3.9.1) or BKEY (2:6.22.1), respectively. Its main use is as a utility for the customized procedures BCIDENTIFY (2:6.21.4), BRPREDICT (2:3.9.4) and BKIDENTIFY (2:6.22.3).

The characteristics of the specimens are specified using the X and VALUES parameters. Each X setting should be one of the factors or variates in the tree, and the corresponding VALUES setting should be a scalar, variate or text defining its values for the specimens. If X is a variate, VALUES may be a scalar if all the specimens have the same x-value (or if there is only one specimen); it will be a variate if there are several specimens with different x-values. VALUES can be also be a scalar or variate if X is a factor. Alternatively, VALUES may be a text (with one or several values) if the factor X has labels. Any restrictions on X or VALUES are ignored.

The tree is supplied by the TREE option. The TESTELEMENT option indicates which element of the pointer of information, stored at each node of the tree, contains the test to be done there. For trees constructed by procedures BCLASSIFICATION, BREGRESSION or BKEY the test element is the second element of the pointers. In trees constructed by BKEY the test is a factor whose (ordinal) level number defines the branch to take from the node. Alternatively, the tests in trees constructed by BCLASSIFICATION and BREGRESSION are simple logical expressions like

```
    X < 1
```

or

```
    X .IN. !t(red,blue)
```

where a "true" result selects the first branch, and a "false" result selects the second. BIDENTIFY allows for expressions containing a single relational operator from the following list:

equality	.EQ. or ==
string equality	.EQS.
non-equality	.NE. or /= or <>

string non-equality	`.NES.`
less than	`.LT.` or `<`
less than or equals	`.LE.` or `<=`
greater than	`.GT.` or `>`
greater than or equals	`.GE.` or `>=`
inclusion	`.IN.`
non-inclusion	`.NI.`

If the factor or variate in the test is not in the list supplied by the `X` parameter, all the branches from the node must be followed, and the specimen will reach several terminal nodes. All the branches must also be taken if the specimen has a missing value for the `X` variable in the test.

The `TERMINALNODES` option saves the numbers of the terminal nodes that the specimens reach in the tree. If there is a single specimen, `TERMINALNODES` will be a scalar or a variate. If there are several specimens, it will be a pointer of scalars or variates.

4.13 Numerical algorithms: the NAG directive

NAG directive

Calls an algorithm from the NAG Library.

Options

`PRINT` = *string token*	Controls printed output (`algorithms`, `monitoring`); default `*` i.e. none
`NAME` = *string token*	Name of the algorithm to call; default `*` i.e. none
`ZDZ` = *string token*	Value to be given to zero divided by zero in Genstat expressions defined in the `ARGUMENTS` (`missing`, `zero`); default `miss`
`TOLERANCE` = *scalar*	If the scalar is non missing, this defines the smallest non-zero number for use in Genstat expressions defined in the `ARGUMENTS`; otherwise it accesses the default value, which is defined automatically for the computer concerned
`SEED` = *scalar*	Seed to use for any random number generation in Genstat expressions defined in the `ARGUMENTS`; default 0
`INDEX` = *scalar*	If a Genstat expression defined in the `ARGUMENTS` has a list of structures before the assignment operator (=), the scalar indicates the position within the list of the structure currently being evaluated

Parameters

`ARGUMENTS` = *pointer*	Arguments for the call
`RESULT` = *scalar*	Stores the result for algorithms that take the form of a function rather than a subroutine

`NAG` provides access to some specific algorithms in the Numerical Algorithms Group's subroutine libraries. You can set option `PRINT=algorithms` to list those that are currently available. The other setting `monitoring` gives additional monitoring from algorithms like `D02KDF` that can give additional monitoring information from a `MONIT` subroutine. (`NAG` includes a custom version of `MONIT` for each routine, that provides all the relevant information.)

The name of the algorithm is specified using the `NAME` option. It is best to give the name in full, as the NAG names may not be distinct in their first four characters and so the standard abbreviation rules (e.g. that four characters are sufficient) cannot be guaranteed in all future releases. The arguments for the call are supplied, in a pointer, using the ARGUMENTS parameter. These must be in the order required by the algorithm, and input arguments must be of the correct type (number or string) and shape (vector, matrix and so on); for details see the relevant NAG documentation. Output arguments are defined automatically from the results. The RESULT parameter saves the result if the NAG algorithm is a function rather than a subroutine.

Example 4.13a uses NAG routine G12BAF to fit a Cox proportional hazards model. The arguments of G12BAF (from the NAG documentation; see e.g. www.nag.co.uk) are as follows.

1: OFFSET – character*1, Input	If OFFSET = 'Y', an offset must be included in OMEGA. If OFFSET = 'N', no offset is included in the model. Constraint: OFFSET = 'Y' or 'N'.
2: N – integer, Input	The number of data points, *n*. Constraint: N ≥ 2.
3: M – integer, Input	The number of covariates in array Z. Constraint: M ≥ 1.
4: NS – integer, Input	The number of strata. If NS > 0 then the stratum for each observation must be supplied in ISI. Constraint: NS ≥ 0. (Note: *strata* here means groups as in stratified surveys, not error strata as in ANOVA; 2:4.2).
5: Z(LDZ,M) – real array, Input	The *i*th row must contain the covariates that are associated with the *i*th failure time given in T.
6: LDZ – integer, Input	The size of the first dimension of the array Z. Constraint: LDZ ≥ N.
7: ISZ(M) – integer array, Input	Indicates which subset of covariates is to be included in the model. If ISZ(*j*) ≥ 1, the *j*th covariate is included in the model. If ISZ(*j*) = 0, the *j*th covariate is excluded from the model and not referenced. Constraints: ISZ(*j*) ≥ 0 and at least one and at most n_0 – 1 elements of ISZ must be non-zero, where n_0 is the number of observations excluding any with zero value of ISI.
8: IP – integer, Input	The number of covariates included in the model as indicated by ISZ. Constraint: IP = number of non-zero values of ISZ.
9: T(N) – real array, Input	The vector of *n* failure censoring times.
10: IC(N) – integer array, Input	The status of the individual at time *t* given in T. If IC(i) = 0, the *i*th individual has failed at time T(*i*). If IC(*i*) = 1, the *i*th individual has been censored at time T(*i*). Constraint: IC(*i*) = 0 or 1 for *i* = 1, 2 ... N.
11: OMEGA(*) – real array, Input	Supplies the offset variate when OFFSET = 'Y'. Note: the dimension of the array OMEGA must be at least N if OFFSET = 'Y', and 1 otherwise.
12: ISI(*) – integer array, Input	Supplies the stratum indicators when NS > 0; you can also set the value to 0 to exclude data points from the analysis. Note: the dimension of ISI must be at least N if NS > 0 and 1 otherwise. Constraints: values must lie between 0 and NS, and at least IP values must be greater than 0.
13: DEV – real, Output	Saves the deviance, that is −2 × (maximized log marginal likelihood).
14: B(IP) – real array Input/Output	On entry: initial estimates of the covariate coefficient parameters i.e. B(*j*) must contain the initial estimate of the coefficient of the covariate in Z corresponding to the *j*th

non-zero value of ISZ. On exit: B(*j*) contains the estimate of the coefficient of the covariate stored in the *i*th column of Z where *i* is the *j*th non-zero value in the array ISZ.

15: SE(IP) – real array, Output — Saves the asymptotic standard errors of the estimates contained in B.

16: SC(IP) – real array, Output — Saves the value of the score function for the estimates contained in B.

17: COV(IP*(IP+1)/2) – real array, Output
Saves the variance-covariance matrix of the parameter estimates in B.

18: RES(N) – real array, Output — Saves the residuals.

19: ND – integer, Output — Saves the number of distinct failure times.

20: TP(NDMAX) – real array, Output
Saves the distinct failure times.

21: SUR(NDMAX,*) – real array, Output
If NS = 0, SUR(*i*,1) saves the estimated survival function for the *i*th distinct failure time. If NS > 0, SUR(*i*,*k*) contains the estimated survival function for the *i*th distinct failure time in the *k*th stratum. Note: the second dimension of the array SUR must be at least 1 if NS = 0, and at least NS if NS > 0.

22: NDMAX – integer, Input — The size of first dimension of the array SUR. Constraint: NDMAX ≥ the number of distinct failure times (as returned in ND).

23: TOL – real, Input — Indicates the accuracy required for the estimation. Convergence is assumed when the decrease in deviance is less than TOL × (1.0 + current deviance). This corresponds approximately to an absolute precision if the deviance is small and a relative precision if the deviance is large. Constraint: TOL ≥ 10 × machine precision.

24: MAXIT – integer, Input — The maximum number of iterations to be used for computing the estimates. If MAXIT is set to 0, then the standard errors, score functions, variance-covariance matrix and the survival function are computed for the input values of the coefficients in B but these are not updated. Constraint: MAXIT ≥ 0.

25: IPRINT – integer, Input — Indicates if the printing of information on the iterations is required. If IPRINT = 0, there is no printing, if IPRINT > 0 then the deviance and the current estimates are printed every IPRINT iterations.

26: WK(IP*(IP+9)=2+N) – real array
Workspace

27: IWK(2*N) – integer array — Workspace

28: IFAIL – integer Input/Output — On entry: IFAIL must be set to 0, 1 or –1 to indicate what to do if a fault occurs (see below). On exit: IFAIL = 0 if no faults have occurred;
IFAIL = 1 if OFFSET ≠ 'Y' or 'N', or M < 1, or N < 2, or NS < 0, or LDZ < N, or TOL < 10 × machine precision, or MAXIT < 0;
IFAIL = 2 if ISZ(*i*) < 0 for some *i*, or the value of IP is incompatible with ISZ, or IC(*i*) ≠ 1 or 0, or ISI(*i*) < 0 or

ISI(*i*) > NS, or number of values of ISZ(*i*) > 0 is greater than or equal to n_0, the number of observations excluding any with ISI(*i*) = 0, or all observations are censored (i.e. IC(*i*) = 1 for all *i*), or NDMAX is too small;
IFAIL = 3 if the matrix of second partial derivatives is singular (try different starting values or include fewer covariates);
IFAIL = 4 if overflow has been detected (try using different starting values);
IFAIL = 5 if convergence has not been achieved in MAXIT iterations;
IFAIL = 6 if in the current iteration 10 step halvings have been performed without decreasing the deviance from the previous iteration (convergence is then assumed).

The failure indicator IFAIL occurs in many of the NAG routines (see Chapter P01 of the NAG documentation, e.g. at www.nag.co.uk, for details). The input setting 0 requests a *hard fail*, which halts execution if a fault occurs; 1 requests a silent *soft fail*, which allows execution to continue and does not generate an error message; and -1 requests a noisy *soft fail*, which allows execution to continue but does generate an error message.

The example fits a Cox proportional hazards model to some data on remission times of child patients with acute leukaemia (see Gehan 1965). There are two treatment groups: one treated with 6-mercaptopurine and the other acting as a control. Lines 2-21 define Genstat structures for each arguments of G12BAF. The character argument OFFSET is represented by a text with a single string of one character. Single integer and real values, like N or DEV, are both represented by scalars. Integer and real arrays, like IC or B, are represented by variates. However, the array COV which is essentially a symmetric matrix, is actually a symmetric matrix in Genstat. Data structures need to be defined for the workspace arguments, but NAG can set up their values automatically with the required lengths. It can also define the output structures automatically. Lines 22-24 define a pointer Args containing the 28 Genstat structures in the order defined for the 28 arguments of G12BAF. Line 25 calls G12BAF, and lines 26 and 27 print some of the output arguments.

Example 4.13a

```
 2  VARIATE    [VALUES=1,1,2,2,3,4,4,5,5,8,8,8,8,11,11,12,12,15,17,22,23,\
 3             6,6,6,6,7,9,10,10,11,13,16,17,19,20,22,23,25,32,32,34,35] Time
 4  &          [VALUES=24(0),1,0,1,0,1,1,0,0,1,1,1,0,0,1,1,1,1,1] Censor
 5  FACTOR     [LABELS=!t(control,'6-mercaptopurine'); VALUES=21(1,2)] Treat
 6  TEXT       [VALUE='N'] OFFSET
 7  VARIATE    T,IC; VALUE=Time,Censor
 8  CALCULATE  N = NVALUES(T)
 9  SCALAR     M,NS; VALUE=1,0
10  MATRIX     [ROWS=N; COLUMNS=M] Z
11  CALCULATE  Z$[*;1] = Treat .EQ. 2
12  CALCULATE  LDZ = NVALUES(Z)
13  VARIATE    [VALUES=1] ISZ
14  CALCULATE  IP = SUM(ISZ)
15  VARIATE    [VALUES=1] OMEGA,ISI
16  VARIATE    [NVALUES=IP] SE,SC
17  VARIATE    [VALUES=0] B
18  SYMMETRIC  [ROWS=IP] COV
19  VARIATE    [NVALUES=N] RES
20  SCALAR     TOL,MAXIT,IPRINT; VALUE=1.E-6,50,0
21  VARIATE    WK,IWK
22  POINTER    [VALUES=OFFSET,N,M,NS,Z,LDZ,ISZ,IP,T,\
23             IC,OMEGA,ISI,DEV,B,SE,SC,COV,RES,ND,\
24             TP,SUR,NDMAX,TOL,MAXIT,IPRINT,WK,IWK,IFAIL] Args
25  NAG        [NAME=G12BAF] Args
26  PRINT      DEV,ND,IFAIL
```

```
        DEV          ND       IFAIL
      172.8       17.00           0

27  PRINT       B,SE

          B          SE
     -1.509      0.4096
```

Next, in Example 4.13b, we use NAG routine `D01AMF` to integrate the function
$1 / ((x+1) \times \sqrt{x})$
from zero to infinity (the answer should be π). The arguments of `D01AMF` are as follows.

1: `F` – real function	An external function `F(`*x*`)` that returns the value of the integrand *f* at a given point *x*.
2: `BOUND` – real Input	The finite limit of the integration range (if present); `BOUND` is not used if the interval is doubly infinite.
3: `INF` – integer Input	indicates the kind of integration range: if `INF` = 1, the range is [`BOUND`, ∞); if `INF` = −1, the range is (−∞, `BOUND`]; if `INF` = 2, the range is (−∞, ∞). Constraint: `INF` = −1, 1 or 2.
4: `EPSABS` – real Input	The absolute accuracy required; if `EPSABS` is negative, the absolute value is used.
5: `EPSREL` – real Input	The relative accuracy required; if `EPSREL` is negative, the absolute value is used.
6: `RESULT` – real Output	The approximation to the integral.
7: `ABSERR` – real Output	An estimate of the modulus of the absolute error, which should be an upper bound for the absolute difference between the integral and `RESULT`.
8: `W(LW)` – real array Output	Details of the computation (end-points of the sub-intervals used by `D01AMF` along with the integral contributions and error estimates over these sub-intervals).
9: `LW` – integer Input	The size of the array `W`. The value of `LW` (together with that of `LIW` below) imposes a bound on the number of subintervals into which the interval of integration may be divided by the routine. The number of subintervals cannot exceed `LW` / 4. The more difficult the integrand, the larger `LW` should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems. Constraint: `LW` ≥ 4.
10: `IW(LIW)` – integer array Output	`IW(1)` contains the actual number of sub-intervals used. The rest of the array is used as workspace.
11: `LIW` – integer Input	The size of the array `IW`. The number of sub-intervals into which the interval of integration may be divided cannot exceed `LIW`. Suggested value: `LIW` = `LW` / 4. Constraint: `LIW` ≥ 1.
12: `IFAIL` – integer Input/Output	On entry: `IFAIL` must be set to 0, 1 or −1 to indicate what to do if a fault occurs (see below). On exit: `IFAIL` = 0 if no faults have occurred; `IFAIL` = 1 if the maximum number of subdivisions allowed with the given workspace has been reached, without the requested accuracy requirements being achieved; `IFAIL` = 2 if round-off error prevents the requested

tolerance from being achieved – the error may be underestimated (consider requesting less accuracy);

`IFAIL` = 3 if extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval;

`IFAIL` = 4 if the requested tolerance cannot be achieved, because the extrapolation does not increase the accuracy satisfactorily – the returned result is the best which can be obtained;

`IFAIL` = 5 suggests that the integral is probably divergent, or slowly convergent (it must be noted that divergence can also occur with any other non-zero value of `IFAIL`);

`IFAIL` = 6 if on entry, `LW` < 4, or `LIW` < 1, or `INF` ≠ −1, 1 or 2.

The first argument for `D01AMF` is an external Fortran function to calculate the function f(x) to be integrated, at a supplied value of x. An argument like this is specified for the `NAG` directive by supplying a pointer whose first element is a Genstat expression, or a pointer to several Genstat expressions, to do the necessary calculations. If the argument is an external function, the next element of the pointer should be the Genstat data structure that receives the result of the calculation in the expression(s); if it is a subroutine, this element is omitted. The remaining elements should be the Genstat data structures that correspond to the arguments of the external function or subroutine, in the order in which they occur in the definition of the function or subroutine in the NAG documentation. So, in line 2, we define the expression `Func` to do the calculation. Then, in line 3, the pointer `F` is defined with `Func` as its first element, the result `Y` of the expression in `Func` as its next element, and the argument `X` of the expression as its remaining element.

The expression or expressions are evaluated within the NAG directive by making a call to the `CALCULATE` directive. The `ZDZ`, `TOLERANCE`, `SEED` and `INDEX` options of the `NAG` directive can be used to set the corresponding options of `CALCULATE` for the call.

Lines 4 and 5 define Genstat data structures for the other input arguments of `D01AMF`, and lines 6 and 7 put them into the pointer `Args`. Line 8 calls `D01AMF`, and lines 8-9 print some of the results. Notice that the output arguments have been defined by default, and that the integer output argument `IFAIL` has been defined with a decimals attribute (as set by the `DECIMALS` parameter of the `SCALAR` directive; 2.2.1) of zero, so that it prints by default as an exact integer.

Example 4.13b

```
   2  EXPRESSION [VALUE=( Y = 1 / ((X+1)*SQRT(X)) )] Func
   3  POINTER    [VALUES=Func,Y,X] F
   4  SCALAR     BOUND,INF,EPSABS,EPSREL,LW; VALUE=0,1,0,0.0001,800
   5  SCALAR     LIW; VALUE=LW/4
   6  POINTER    [VALUES=F,BOUND,INF,EPSABS,EPSREL,RESULT,ABSERR,\
   7             W,LW,IW,LIW,IFAIL] Args
   8  NAG        [NAME=D01AMF] Args
   9  PRINT      RESULT; DECIMALS=6

      RESULT
    3.141593

  10  PRINT      ABSERR,IFAIL

      ABSERR       IFAIL
  0.00002651           0
```

5 Programming in Genstat

The commands that you input to Genstat are known as a Genstat *program*. This consists of a sequence of one or more jobs. The first job starts automatically at the start of the program. Later, if you want, you can begin a subsequent job using the `JOB` and `ENDJOB` directives. The effect is equivalent to restarting Genstat (data structures are deleted, the graphics environment is reset, and so on) except that any files that have been attached to Genstat retain their current status. So, for example, Genstat will continue to add output to the end of an output file, and will continue reading from the current point of an input file.

`JOB`	starts a Genstat job, ending the previous one if necessary (5.1.1)
`ENDJOB`	ends a job (5.1.2)

The whole program is terminated by a `STOP` directive:

`STOP`	ends a Genstat program (5.1.3)

Statements within a program can be repeated using a `FOR` loop. The loop is introduced by a `FOR` statement. This is followed by the series of statements that is to repeated (that is, the contents of the loop), and the end of the loop is marked by an `ENDFOR` statement. Parameters of the `FOR` directive allow lists of data structures to be specified so that the statements in the loop operate on different structures each time that it is executed.

`FOR`	indicates the start of a loop (5.2.1)
`ENDFOR`	marks the end of a loop (5.2.1)

Genstat has two ways of choosing between sets of statements. The block-if structure consists of one or more alternative sets of statements. The first set is introduced by an `IF` statement. There may then be further sets introduced by `ELSIF` statements. Then there may be a final set introduced by an `ELSE` statement, and the whole structure is terminated by an `ENDIF` structure. The `IF` statement, and each `ELSIF` statement, contains a single-valued logical expression. Genstat evaluates each one in turn and executes the statements following the first `TRUE` logical found; if none of them is true, Genstat executes the statements following the `ELSE` statement (if any).

`IF`	introduces a block-if structure (5.2.2)
`ELSIF`	introduces an alternative set of statements in a block-if structure (5.2.2)
`ELSE`	introduces a default set of statements for a block-if structure (5.2.2)
`ENDIF`	marks the end of a block-if structure (5.2.2)

The multiple-selection structure consists of several sets of statements. The first is introduced by a `CASE` statement. Subsequent sets are introduced by `OR` statements. There can then be a final, default, set introduced by an `ELSE` statement, and the end of the structure is indicated by an `ENDCASE` statement. The parameter of the `CASE` statement is an expression which must produce a single number. Genstat rounds this to the nearest integer, *n* say, and then executes the *n*th set of statements. If there is no *n*th set, the statements following the `ELSE` statement are executed (if any).

`CASE`	introduces a multiple-selection structure (5.2.3)
`OR`	introduces an alternative set of statements for a multiple-selection structure (5.2.3)
`ELSE`	introduces a default set of statements for a multiple-selection structure (5.2.3)
`ENDCASE`	marks the end of a multiple-selection structure (5.2.3)

Any control structure (job, block-if structure, loop, multiple-selection structure or procedure – see below) can be abandoned using an `EXIT` statement.

`EXIT`	exits from a control structure (5.2.4)

Sequences of statements can be formed into Genstat procedures. This not only makes them simpler for you to use; it also means that you can make them easily available to other users. The use of a procedure looks just like one of the Genstat directives, with its own options and parameters, which transfer information to and from the procedure. Otherwise the procedure is completely self-contained. There is a standard, officially-supported procedure library, which is automatically available whenever you run Genstat. Details are available on-line from the procedures in the `help` module of the library. You can also write your own procedures (5.3.2), and form your own libraries with their own on-line help (5.3.4).

`LIBHELP`	provides help information for Library procedures (5.3.1)
`LIBEXAMPLE`	accesses examples and source code of Library procedures (5.3.1)
`LIBVERSION`	provides the name of the current Genstat Procedure Library (5.3.1)
`NOTICE`	provides information about Genstat, including instructions for authors of Library procedures

The start of a procedure is indicated by a `PROCEDURE` statement. Then `OPTION` and `PARAMETER` statements can be given to define the arguments of the procedure. These are followed by the statements to be executed when the procedure is called, terminated by an `ENDPROCEDURE` statement.

`PROCEDURE`	introduces a procedure, and defines its name (5.3.2)
`OPTION`	defines the options of a procedure (5.3.2)
`PARAMETER`	defines the parameters of a procedure (5.3.2)
`CALLS`	lists the procedures called by a procedure (5.3.2)
`ENDPROCEDURE`	indicates the end of a procedure (5.3.2)

Commands are available to enable procedure writers to provide their own error handing, to define and access private data structures, to execute macros and to increment counters. You can also discover whether and how a particular command has been implemented.

`FAULT`	checks whether to issue a diagnostic, i.e. a fault, warning or message (5.4.1)
`DISPLAY`	repeats the last Genstat diagnostic (5.4.1)
`WORKSPACE`	accesses "private" data structures for use in procedures (5.4.2)
`EXECUTE`	executes the statements contained within a text (5.4.3)
`COUNTER`	increments a multi-digit counter using non base-10 arithmetic (5.4.4)
`COMMANDINFORMATION`	provides information about whether (and how) a command has been implemented (5.4.5)
`SPSYNTAX`	puts details about the syntax of commands into a spreadsheet
`SYNTAX`	obtains details about the syntax of a command (5.4.6)

Genstat has commands to help you debug your programs. The execution of any control structure (job, block-if structure, loop, multiple-selection structure or procedure) can be interrupted explicitly (so that you can enter other commands such as `PRINT`) using a `BREAK` statement, or implicitly by using `DEBUG`; once `DEBUG` has been entered, Genstat will produce breaks automatically at regular intervals, until it meets an `ENDDEBUG` statement.

`BREAK`	suspends the execution of a control structure (5.5.1)
`ENDBREAK`	continues execution of a control structure, following a break (5.5.1)
`DEBUG`	can cause a break to take place after the current statement (and at specified intervals thereafter), or immediately after the next fault (5.5.2)
`ENDDEBUG`	cancels `DEBUG` (5.5.2)

You can modify aspects of the "environment" of the current Genstat job, such as whether or not Genstat starts output from a statistical analysis at the top of a new page, or whether it should pause during interactive output. You can also copy details of these environmental settings into Genstat data structures so that, for example, you can react appropriately within a procedure. User-defined defaults can be specified for the options and parameters of any directive or procedure.

`SET`	sets details of the "environment" of a Genstat job (5.6.1)
`GET`	accesses information about the Genstat environment (5.6.2)
`SETOPTION`	sets or modifies defaults of options of Genstat directives or procedures (5.6.3)
`SETPARAMETER`	sets or modifies defaults of parameters of Genstat directives or procedures (5.6.3)

In many implementations of Genstat, you can suspend the execution of Genstat and return to the operating system of the computer to execute commands, for example to list or edit files on the computer. Likewise, it may be possible to halt the execution of Genstat to execute some other computer program. Some implementations also allow you to incorporate your own programs into Genstat. The `OWN` directive calls a subroutine called `OWN`, within the Fortran code of Genstat, which may be modified to call the program. The new code must then be recompiled and linked into a new version of Genstat.

`SUSPEND`	suspends the execution of Genstat to carry out operating-system commands (5.7.1)
`PASS`	runs another computer program, taking data from Genstat and transferring results back (5.7.2)
`EXTERNAL`	declares an external function in a DLL for use by the `OWN` function (5.7.3)
`OWN`	executes the user's own code linked into Genstat

5.1 Genstat programs

A Genstat program is a sequence of statements involving either standard Genstat directives or procedures (1.1). You may often wish to examine several different sets of data within the same program: for example, you may want to be able to collect several analyses in one batch run (1.1.2), or to be able to end one analysis and start a different one, with different data, when you are running Genstat interactively (1.1.1).

The `JOB` and `ENDJOB` directives can be used to partition a Genstat program into separate *jobs*. A job is a self-contained subsection of a program. All data structures and procedures are lost at the end of each job. Any setting defined by a `UNITS` statement (2.3.4) is deleted, as are the special structures set up by analyses like regression and analysis of variance (2.9). The graphics environment is also reset to the initial default. Thus, in many ways, it is as though Genstat was starting again for each new job.

However, any files that have been attached to Genstat retain their current status from job to job. So, for example, Genstat will continue to add output to the end of an output file, or will continue reading from the current point of an input file.

The JOB directive also has options that allow you to modify some aspects of the Genstat environment: for example what prompt will be used for input and whether input lines are reprinted in an output file. The default settings of the options will leave these aspects unchanged so, if any aspect is modified, it will remain in that form (unless modified again) in any subsequent job. The initial settings, which apply at the outset of a program, are described in 5.1.1; however, remember that it is possible to arrange for Genstat to run commands from a *start-up* file (5.6.4) before it executes the first statement of a program, so the initial environment can differ from machine to machine.

5.1.1 The JOB directive

JOB directive

Starts a Genstat job.

Options

INPRINT = *string tokens*	Printing of input as in PRINT option of INPUT (statements, macros, procedures, unchanged); default unch
OUTPRINT = *string tokens*	Additions to output as in PRINT option of OUTPUT (dots, page, unchanged); default unch
DIAGNOSTIC = *string tokens*	Defines the least serious class of Genstat diagnostic which should still be generated (messages, warnings, faults, extra, unchanged); default unch
ERRORS = *scalar*	Limit on number of error diagnostics that may occur before the job is abandoned; default * i.e. no change
PROMPT = *text*	Characters to be printed for the input prompt
WORDLENGTH = *string token*	Length of word (8 or 32 characters) to check in identifiers, directives, options, parameters and procedures (long, short); default * i.e. no change

Parameter

text	Name to identify the job

The JOB directive is used to start a new job. It has a parameter which can be set to a text to identify the job (for example in the message at the end of the job), and options to control some aspects of the Genstat "environment". However, Genstat will automatically start a job at the beginning of a program, or after an ENDJOB statement, so you do not need to give a JOB statement unless you wish to define an identifying text or to modify the environment.

The INPRINT option specifies which pieces of input from the current input channel will be recorded in the current output file. (The current input channel may be a file or, in an interactive run, it may also be the keyboard.) The settings correspond to three types of input:

statements	statements that are typed explicitly on the keyboard or which occur explicitly in an input file,
macros	statements or parts of statements that have been supplied in macros, using the ## notation (1.8.2), and
procedures	statements occurring within procedures.

The initial default is to record only statements for input from a file, or to record nothing if input is from the keyboard. The recording of input can be modified also by the INPRINT option of the SET directive (5.6.1), or by the PRINT option of INPUT (3.4.1).

The OUTPRINT option controls the way in which the output from many Genstat directives will start: page ensures that output to a file will start at the head of a page, and dots produces a line of dots beginning with the line number of the statement that has generated the analysis. The initial default is to give a new page and a line of dots if output is to a file, but neither if output is to the screen. This can be modified also by the OUTPRINT option of the SET directive (5.6.1), or by the PRINT option of OUTPUT (3.4.3).

The DIAGNOSTICS option controls the reporting of errors and possible mistakes. In order of increasing seriousness there three classes of diagnostic: messages, warnings and faults. Messages are comments that are made to draw your attention to things that might need closer investigation, like large residuals in an analysis of variance or a regression. Warnings are definite errors, but ones that are not sufficiently serious to prevent Genstat from continuing; an example would be an attempt to print a data structure with no values. Faults are the most serious type of error. A fault in a batch run will cause Genstat to stop executing the current job. However, Genstat will continue to read and interpret the statements so that it can find the start of the next job (if any); at the same time it will report any further errors that it finds, up to the number specified by the ERRORS option. The setting of DIAGNOSTICS indicates the level of stringency to be adopted. Thus, if DIAGNOSTICS=warnings, Genstat will report faults and warnings (but not messages), while DIAGNOSTICS=messages ensures that all three classes are reported. The setting extra is similar to messages but will also generate a dump of system information (2.11.2) after any fault. You can prevent the output of any diagnostics by putting DIAGNOSTICS=*. The initial default is to set DIAGNOSTICS=messages. This can be modified also by the DIAGNOSTIC option of the SET directive (5.6.1).

The WORDLENGTH parameter controls the number of characters that are stored and checked in identifiers and names of directives, procedures, options, parameters and functions. In releases prior to 4.2 this was always eight, but from 4.2 onwards you can choose between eight (WORDLENGTH=short) and 32 (WORDLENGTH=long). The initial default is long. This can be modified also by the WORDLENGTH option of the SET directive (5.6.1)

5.1.2 The ENDJOB directive

ENDJOB directive

Ends a Genstat job.

No options or parameters

The ENDJOB directive terminates a job, printing a message summarizing how much workspace has been used. For example:

Example 5.1.2

```
   2  JOB 'Example of ENDJOB message'
   3  PRINT 'This job just prints this message.'

 This job just prints this message.

   4  ENDJOB

******** End of Example of ENDJOB message.
```

You do not need to give an ENDJOB statement before a JOB statement, as JOB will automatically end any existing job before it starts another. Thus you can begin a new job by specifying either JOB or ENDJOB (or both).

5.1.3 The STOP directive

STOP directive

Ends a Genstat program.

No options or parameters

The STOP directive indicates the end of a Genstat program, thus telling the computer that you have finished using Genstat. It also ends the existing job, so there is no need to give an ENDJOB statement beforehand. Any input that follows a STOP statement is ignored.

5.2 Program control

Usually the statements in a Genstat job are executed in sequence, until either ENDJOB or STOP is reached. But, as with most programming languages, you may sometimes want to control the order in which the statements are executed.

If you have several sets of data that are all to be analysed in the same way, you may want to repeat the necessary series of statements for each set. You can do this by preceding the series with a FOR statement, and ending it with an ENDFOR statement. The FOR directive also allows you to specify dummy structures (2.2.2) which point in turn to the data structures of the successive sets.

To be able to write general programs, you may need to be able to choose between alternative sets of statements, according to the exact form of a particular set of data. There are two ways in which you can do this. The directives IF, ELSIF, ELSE and ENDIF allow you to define *block-if* structures (5.2.2). Alternatively, the directives CASE, OR, ELSE and ENDCASE allow you to choose between sets of statements according to an integer value (5.2.3).

The EXIT directive (5.2.4) allows you to abandon any of these control structures while the program is being executed. The exit can be dependent on a condition, for example on an invalid data value or even on a Genstat diagnostic. The Genstat language is designed in accordance with the principles of structured programming: there is no way of "labelling" a statement and no equivalent of the Fortran "GO TO" construct.

5.2.1 FOR loops

FOR directive

Introduces a loop; subsequent statements define the contents of the loop, which is terminated by the directive ENDFOR.

Options

NTIMES = *scalar*	Number of times to execute the loop; default is to execute as many times as the length of the first parameter list or once if the first list is null
INDEX = *scalar*	Records the number of the current time that the loop is being executed
START = *scalar*	Defines an integer initial value for the loop index; default 1

`END` = *scalar*	Defines an integer final value for the loop index
`STEP` = *scalar*	Defines an integer amount by which to increase the index each time the loop is executed; default 1
`VALUES` = *variate*	Defines a set of values to be taken successively by the loop index (overides `START`, `END` and `STEP` if these are specified too)

Parameters

	Any number of parameter settings of the form *identifier = list of data structures*; the *identifier* is set up as a dummy which is then used within the loop to refer, in turn, to the structures in the list

`ENDFOR` directive

Indicates the end of the contents of a loop.

No options or parameters

The `FOR` loop is a series of statements, or a *block*, that is repeated several times. The `FOR` directive introduces the loop and indicates how many times it is to be executed. In its simplest form `FOR` has no parameters, and the number of times is indicated by the `NTIMES` option. Thus the iterative calculation of a square root in the example in 1.8.2 can be specified in a `FOR` loop like this:

```
FOR [NTIMES=3; INDEX=Iteration]
  CALCULATE Previous = Root
  & Root = (X/Previous + Previous)/2
  PRINT Iteration,Root,Previous; DECIMALS=0,4,4
ENDFOR
```

The sequence of `CALCULATE` and `PRINT` statements is repeated three times (exactly as in 1.8.2).

The `INDEX` option allows you to record the loop index in a scalar. By default this is the number of the time that the loop is currently being executed. So, in the statement below, the index Count will take the values 1, 2 and 3.

```
FOR [NTIMES=3; INDEX=Count]
```

The options `START`, `END` and `STEP` allow you to define a loop index that does not start at one, and does not increase by one each time the loop is executed. They should all be set to integers; any non-integer value is rounded to the nearest integer. (Integer calculations are exact, so this avoids inaccuracies due to numerical round-off when loops are executed many times.) `START` specifies the `INDEX` value on the first time that the loop is executed (default 1). `STEP` defines how it changes between one time that the loop is executed and the next (default 1). So, for example, on the second time `INDEX` will be `START + STEP`. `END` provides an alternative way of specifying how many times to execute the loop – it stops when the next index will go beyond `END`. For example, the statement below

```
FOR [INDEX=Count; START=3; END=8; STEP=2]
```

defines a loop that will be executed three times, with the index variable `Count` taking the values 3, 5 and 7; the next value would be 9, which goes beyond 8. The default `STEP` is one. `STEP` can also be negative. So, this statement

```
FOR [INDEX=Count; START=3; END=-4; STEP=-2]
```

defines a loop that will be executed four times, with the index variable Count taking the values 3, 1, -1 and -3; the next value would be -5, which goes beyond -4. If you specify `NTIMES` as well as `END`, they must both define the same number of times to execute the loop.

The VALUES option allows you to specify an arbitrary sequence of values for the loop index, and these need not be integers. The setting is a variate. So, for example, here

```
VARIATE [VALUES=0, 0.5, 1, 1.5, 2, 1.5, 1, 0.5, 0] Cvals
FOR [INDEX=Count; VALUES=Cvals]
```

Count will first increase from 0 to 2 in steps of 0.5, and then decrease back down to 0. The number of values in the VALUES variate must be the same as the value supplied by NTIMES if both options are specified. VALUES overrides START, END and STEP if these are specified too.

The INDEX is defined automatically as a scalar if it has not already been declared. If VALUES is set, its default number of decimals is set to be the same as the number defined for the VALUES variate (see the DECIMALS parameter of the VARIATE and SCALAR directives), or to take the default number if no decimals have been defined for VALUES. Otherwise the default number of decimals is set to zero.

The parameters of FOR allow you to write a loop whose contents apply to different data structures each time it is executed. Unlike other directives, the parameter names of FOR are not fixed for you by Genstat: you can put any valid identifier before each equals sign. Each of these then refers to a Genstat dummy structure, as described in 2.2.2; so you must not have declared them already as any other type of structure. The first time that the loop is executed, they each point to the first data structure in their respective lists, next time it is the second structure, and so on. The list of the first parameter must be the longest; other lists are recycled as necessary.

If you specify parameters you do not need to specify NTIMES but, if you specify both the value of NTIMES must be the same as the length of the first parameter list.

You can specify as many parameters as you need. For example

```
FOR Ind=Age,Name,Salary; Dir='descending','ascending'
  SORT [INDEX=Ind; DIRECTION=#Dir] Name,Age,Salary
  PRINT Name,Age,Salary
ENDFOR
```

is equivalent to the sequence of statements

```
SORT [INDEX=Age; DIRECTION='descending'] Name,Age,Salary
PRINT Name,Age,Salary
SORT [INDEX=Name; DIRECTION='ascending'] Name,Age,Salary
PRINT Name,Age,Salary
SORT [INDEX=Salary; DIRECTION='descending'] Name,Age,Salary
PRINT Name,Age,Salary
```

printing the units of the text Name, and variates Age and Salary, first in order of descending ages, then in alphabetic order of names, and finally in order of descending salaries.

You can put other control structures inside the loop. So, for example, you can have loops within loops.

When you are using loops interactively, you may find it helpful to use the PAUSE option of SET to request Genstat to pause after every so many lines of output (5.6.1). Another useful directive is BREAK, which specifies an explicit break in the execution of the loop (5.5.1).

5.2.2 Block-if structures

The component parts of a *block-if* structure are delimited by IF, ELSIF, ELSE and ENDIF statements.

IF directive

Introduces a block-if control structure.

No options

Parameter

expression	Logical expression, indicating whether or not to execute the first set of statements.

`ELSIF` directive

Introduces a set of alternative statements in a block-if control structure.

No options

Parameter

expression	Logical expression to indicate whether or not the set of statements is to be executed.

`ELSE` directive

Introduces the default set of statements in block-if or in multiple-selection control structures.

No options or parameters

`ENDIF` directive

Indicates the end of a block-if control structure.

No options or parameters

A *block-if* structure consists of one or more alternative sets of statements. The first of these is introduced by an `IF` statement. There may then be further sets introduced by `ELSIF` statements. Then you can have a final set introduced by an `ELSE` statement, and the whole structure is terminated by an `ENDIF` statement. Thus the general form is:
first

```
IF expression
  statements
```

then either none, one, or several blocks of statements of the form

```
ELSIF expression
  statements
```

then, if required, a block of the form

```
ELSE
  statements
```

and finally the statement

```
ENDIF
```

Each expression must evaluate to a single number, which is treated as a logical value: a zero or missing value is treated as *false* and non-zero as *true* (4.1.1). Genstat executes the block of statements following the first true expression. If none of the expressions is *true*, the block of statements following `ELSE` (if present) is executed.

You can thus use these directives to built constructs of increasing complexity. The simplest form would be to have just an `IF` statement, then some statements to execute, and then an `ENDIF`. For example:

```
IF MINIMUM(Sales) < 0
  PRINT 'Incorrect value recorded for Sales.'
ENDIF
```

If the variate `Sales` contains a negative value, the `PRINT` statement will be executed. Otherwise

Genstat goes straight to the statement after ENDIF.

To specify two alternative sets of statements, you can include an ELSE block. For example

```
IF Age < 21
  CALCULATE Pay = Hours*3.25
ELSE
  CALCULATE Pay = Hours*4.5
ENDIF
```

calculates Pay using two different rates: 3.25 for Age less than 21, and 4.5 otherwise.

Finally, to have several alternative sets, you can include further sets introduced by ELSIF statements. Suppose that we want to assign values to X according to the rules:

X=1 if Y=1
X=2 if Y ≠ 1 and Z=1
X=3 if Y ≠ 1 and Z=2
X=4 if Y ≠ 1 and Z ≠ 1 or 2

This can be written in Genstat as follows:

```
IF Y == 1
  CALCULATE X = 1
ELSIF Z == 1
  CALCULATE X = 2
ELSIF Z == 2
  CALCULATE X = 3
ELSE
  CALCULATE X = 4
ENDIF
```

If Y is equal to 1, the first CALCULATE statement is executed to set X to 1. If Y is not equal to 1, Genstat does the tests in the ELSIF statements, in turn, until it finds a *true* condition; if none of the conditions is *true*, the CALCULATE statement after ELSE is executed to set X to 4. Thus, for Y=99 and Z=1, Genstat will find that the condition in the IF statement is *false*. It will then test the condition in the first ELSIF statement; this produces a *true* result, so X is set to 2. Genstat then continues with whatever statement follows the ENDIF statement. Block-if structures can be nested to any depth, to give conditional constructs of even greater flexibility.

5.2.3 The multiple-selection control structure

The directives CASE, OR, ELSE and ENDCASE allow you to specify alternative blocks of statements, to be selected according to the value of an expression yielding a single integer value.

CASE directive

Introduces a "multiple-selection" control structure.

No options

Parameter

expression	Expression which is evaluated to an integer, indicating which set of statements to execute

---

OR directive

Introduces a set of alternative statements in a "multiple-selection" control structure.

No options or parameters

`ELSE` directive

Introduces the default set of statements in block-if or in multiple-selection control structures.

No options or parameters

`ENDCASE` directive

Indicates the end of a "multiple-selection" control structure.

No options or parameters

A *multiple-selection* control structure consists of several alternative blocks of statements. The first of these is introduced by a `CASE` statement. This has a single parameter, which is an expression that must yield a single number. Subsequent blocks are each introduced by an `OR` statement. There can then be a final block, introduced by an `ELSE` statement, as in the block-if structure (5.2.2). The whole structure is terminated by an `ENDCASE` statement. Thus the general form is: first

```
CASE expression
  statements
```

then either none, one, or several blocks of statements of the form

```
OR
  statements
```

then, if required, a block of the form

```
ELSE
  statements
```

and finally the statement

```
ENDCASE
```

Genstat rounds the expression in the `CASE` expression to the nearest integer, *k* say, and then executes the *k*th block of statements. If there is no *k*th block present (as for example if *k* is negative) the block of statements following the `ELSE` statement is executed, if there is such a block; otherwise an error diagnostic is given. The next example prints the salient details about each day in the song *The twelve days of Christmas*. The scalar `Day` indicates which day it is.

```
CASE Day
  PRINT 'a partridge in a pear tree'
OR
  PRINT 'two turtle doves and a partridge in a pear tree'
OR
  PRINT 'three French hens, two turtle doves \
    and a partridge in a pear tree'
OR
  PRINT 'four calling birds, three French hens ...'
OR
  PRINT 'five gold rings ...'
OR
  PRINT 'six geese a-laying ...'
OR
  PRINT 'seven swans a-swimming ...'
OR
  PRINT 'eight maids a-milking ...'
OR
  PRINT 'nine drummers drumming ...'
OR
```

```
    PRINT 'ten pipers piping ...'
  OR
    PRINT 'eleven ladies dancing ...'
  OR
    PRINT 'twelve lords a-leaping ...'
  ELSE
    PRINT 'sorry, no delivery today'
  ENDCASE
```

CASE statements can be nested to any depth.

5.2.4 Exit from control structures

Sometimes you may want simply to abandon part of a program: you may be unable to do any further calculations or analyses. For example, if you are examining several subsets of the units, you would wish to abandon the analysis of any subset that turned out to contain no observations. Another example would be if you wanted to abandon the execution of a procedure whenever an error diagnostic has appeared. The EXIT directive allows you to exit from any control structure.

EXIT directive

Exits from a control structure.

Options

NTIMES = *scalar*	Number of control structures, *n*, to exit (if *n* exceeds the number of control structures of the specified type that are currently active, the exit is to the end of the outer one; while for *n* negative, the exit is to the end of the -*n*'th structure in order of execution); default 1
CONTROLSTRUCTURE = *string token*	Type of control structure to exit (job, for, if, case, procedure); default for
REPEAT = *string token*	Whether to go to the next set of parameters on exit from a FOR loop or procedure (yes, no); default no
EXPLANATION = *text*	Text to be printed if the exit takes place; default *

Parameter

expression	Logical expression controlling whether or not an exit takes place

In its simplest form EXIT has no parameter setting, and the exit is unconditional: Genstat will always exit from the control structure or structures concerned. You are most likely to use this as part of an ELSE block of a block-if or multiple-selection structure. For example

```
  IF N.GT.0
    CALCULATE Percent = R * 100 / N
  ELSE
    PRINT [IPRINT=*] 'Incorrect value ',N,'  for N.'
    EXIT [CONTROLSTRUCTURE=procedure]
  ENDIF
```

prints an appropriate warning message for a zero or negative value of N, and then exits from a procedure.

If the warning message is simply a text or string, the EXPLANATION option can be used to print it on exit. For example

```
  EXIT [CONTROLSTRUCTURE=procedure;\
    EXPLANATION='Incorrect value for N.'] \
    N.LE.0
  CALCULATE Percent = R * 100 / N
```

has the same effect except that the actual value of `N` is no longer printed.

The `CONTROLSTRUCTURE` option specifies the type of control structure from which to exit. The default setting is `for`, causing an exit from a `FOR` loop (5.2.1). For the other settings: `if` causes an exit from a block-if structure (5.2.2), `case` exits from a multiple-selection structure (5.2.3), `procedure` exits from a procedure (5.3), and `job` causes the entire job to be abandoned. Sometimes, to exit from one type of control structure, others must be left too. To exit from the procedure in the above example, requires Genstat to exit also from the block-if structure. Generally, Genstat does these nested exits automatically, as required. However, inside a procedure, you can exit only from `FOR` loops and block-if or multiple-selection structures that are within the procedure. You cannot put, for example,

```
EXIT [CONTROLSTRUCTURE=if]
```

within a part of the procedure where there is no block-if in operation, and then expect Genstat to exit both from the procedure and from a block-if structure in the outer program from which the procedure was called. Genstat regards a procedure as a self-contained piece of program.

The `NTIMES` option indicates how many control structures of the specified type to exit from. If you ask Genstat to exit from more structures than are currently in operation in your program, it will exit from as many as it can and then print a warning. If `NTIMES` is set to zero or to missing value no exit takes place. If `NTIMES` is set to a negative value, say -*n*, the exit is to the end of the *n*th structure of the specified type, counting them in the order in which their execution began. Consider this example:

```
FOR I=A[1...3]
  FOR J=B[1...3]
    FOR K=C[1...3]
      FOR L=D[1...3]
        "contents of the inner loop, including:"
        EXIT [NTIMES=Nexit]
        "amongst other statements"
      ENDFOR "end of the loop over D[]"
    ENDFOR "end of the loop over C[]"
  ENDFOR "end of the loop over B[]"
ENDFOR "end of the loop over A[]"
```

If the scalar `Nexit` has the value 2, the exit is to the end of the loop over C[]; so the two exits are from the loop over `D[]` and the loop over `C[]`. But if `Nexit` has the value `-2` the exit is to the end of the loop over `B[]`, as this is the second loop to have been started.

A further possibility when `EXIT` is used within a `FOR` loop is that you can choose either to go right out of the loop and continue by executing the statement immediately after the `ENDFOR` statement, or to go to `ENDFOR` and then repeat the loop with the next set of parameter values. To repeat the loop, you need to set option from one one pass through a loop `REPEAT=yes`. For example, suppose that variates `Height` and `Weight` contain information about children of various ages, ranging from five to 11. The `RESTRICT` statement causes the subsequent `DGRAPH` statement to plot only those units of `Height` and `Weight` where the variate `Age` equals `Ageval` (4.4.1). The `EXIT` statement ensures that the graph is not plotted if there are no units of a particular age; the program then continues with `Ageval` taking the next value in the list.

```
FOR Ageval=5,6,7,8,9,10,11
  RESTRICT Height,Weight; CONDITION=Age.EQ.Ageval
  EXIT [REPEAT=yes] NVALUES(Height).EQ.0
  DGRAPH Y=Height; X=Weight
ENDFOR
```

The `REPEAT` option can also be used within a procedures to ask Genstat to call the procedure with the next set of parameter settings.

The example of the heights and weights of children also illustrates the use of the parameter of `EXIT`, to make the effect conditional. The parameter is an expression which must evaluate to a single number which Genstat interprets as a logical value. If the value is zero, the condition is *false* and no exit takes place; for other values the condition is *true* and the exit takes effect as specified. This is particularly useful for controlling the convergence of iterative processes: for example

```
CALCULATE Clim = X/10000
FOR [NTIMES=999]
  CALCULATE Previous = Root
  & Root = (X/Previous + Previous)/2
  PRINT Root,Previous; DECIMALS=4
  EXIT ABS(Previous-Root) < Clim
ENDFOR
```

will calculate the square root of `X` to four significant figures.

5.3 Procedures

Once you start to write programs for complicated tasks, you may wish to keep them to use again in future. The most convenient way of doing this is to form them into procedures. You may also wish to use procedures written by other people.

The use of a Genstat procedure looks exactly the same as the use of one of the standard Genstat directives. You simply give the name of the procedure, and then specify options and parameters as required.

When Genstat meets a statement with a name that it does not recognize as one of the standard Genstat directives, it first looks to see whether you have a procedure of that name already stored in your program. Then it looks in any procedure library that you may have attached explicitly to your program, taking these in order of their channel numbers (5.3.3). The people that manage your computer can define a special *site* library and arrange for this to be attached to Genstat automatically when it is run. If they have done so, this library will be examined next. Finally Genstat looks in the official Genstat procedure library (5.3.1), which is also attached automatically to your program. After locating the required procedure, Genstat reads it in, if necessary, and then executes it. So you do not have to do any more than you would to use a Genstat directive.

The official library thus allows new facilities to be offered to all users. Or your computer manager can make procedures available that cover the special needs of the users at your site, and these will over-ride any procedures of the same name in the official library. Or you can form your own libraries of the procedures that you find particularly useful, and these will always be taken in preference to procedures in the site or the official library. Note however that a procedure cannot have the same name as any of the Genstat directives (5.3.2).

Information is transferred to and from a procedure only by means of its options and parameters. Otherwise the procedure is completely self-contained. Anyone who uses it does not need to know how the program inside operates, what data structures it contains, nor what directives it uses. The data structures inside the procedure are local to the procedure and cannot be accessed from outside.

The first part of this section describes the Genstat Procedure Library. Later we describe how to write your own procedures, and how to form and access procedure libraries of your own.

5.3.1 The Genstat Procedure Library

The Genstat Procedure Library contains procedures contributed not only by the writers of Genstat but also by knowledgeable Genstat users from many application areas - and countries. It is controlled by an Editorial Board, who check that the procedures are useful and reliable, and maintain standards for the documentation. Guidelines for Authors were published in Genstat Newsletter 20, or can be obtained from within Genstat by setting the `PRINT` option of procedure `NOTICE` to `instructions`: i.e.

```
NOTICE [PRINT=instructions]
```

The other `PRINT` settings include: `errors` for information about how to report errors, `release` for information about most recent Genstat release, and `news` for general Genstat news.

Information about the syntax and ways to use the Library procedures is included in Genstat's on-line help system, in the same format as the information about the Genstat directives. You can also access the relevant topics directly using the `LIBHELP` procedure. For example

```
LIBHELP 'SUBSET'
```

opens the page about procedure `SUBSET`. The Help menu in Genstat *for Windows* has options that allow you to access and run an example for each procedure, or copy its source code into a text window. Alternatively, you can access this information directly by using the `LIBEXAMPLE` procedure. For example,

```
LIBEXAMPLE 'SUBSET'; EXAMPLE=Exsub; SOURCE=Ssub
```

copies the example for `SUBSET` into a text called `Exsub` and the source code into a text called `Ssub`.

5.3.2 Forming a procedure

To write your own procedures, you start by giving a `PROCEDURE` statement.

PROCEDURE directive

Introduces a Genstat procedure.

Options

`PARAMETER` = *string token*	Whether to process the structures in each parameter list of the procedure sequentially using a dummy to store each one in turn, or whether to put them all into a pointer so that the procedure is called only once (`dummy, pointer`); default `dumm`
`RESTORE` = *string tokens*	Which aspects of the Genstat environment to store at the start of the procedure and restore at the end (`inprint`, `outprint`, `outstyle`, `diagnostic`, `errors`, `pause`, `prompt`, `newline`, `case`, `run`, `units`, `blockstructure`, `treatmentstructure`, `covariate`, `asave`, `dsave`, `msave`, `rsave`, `tsave`, `vsave`, `vcomponents`, `seeds`, `captions`, `cmethod`, `actionafterfault`, `unsetdummy`, `all`); default `*`
`SAVE` = *text*	Text to save the contents of the procedure (omitting comments and some spaces)
`WORDLENGTH` = *string token*	Length of word (32 or 8 characters) to check in identifiers, directives, options, parameters and procedures within the procedure (`long`, `short`); default `*` i.e. no change

Parameter

text	Name of the procedure

The `PROCEDURE` directive starts the definition of a procedure. It has a single parameter which defines the name of the procedure. This can be up to 32 characters, and follows the same rules as for the identifiers of data structures: the first character must be a letter, the second to the 32nd can be either letters or digits, and characters beyond the 32nd are ignored. However the name cannot be suffixed, and Genstat will warn you if the first four characters are the same as those of a Genstat directive. If so, you will be unable to abbreviate the name fully (down to as few as four characters), but you will need to give enough characters to distinguish it from the directive. If there is ambiguity in the name of a command, Genstat selects the directive or procedure to use according to the following order of priority: directives, user-defined procedures, procedures in libraries attached by the user (in order of channel number), procedures in the site library, and procedures in the official library.

The PARAMETER option indicates whether the settings in any list specified for the parameters of the procedure are to be taken one at a time, or whether they need to be processed together. The difference between these alternatives can be illustrated by considering some of the Genstat directives. For example, with

```
    ANOVA Height,Weight; RESIDUALS=Hres,Wres
```

Genstat will first do an analysis with the values in the Height variate and store the resulting residuals in the variate Hres; it then analyses Weight and stores the residuals in Wres. This action corresponds to the default setting PARAMETER=dummy; inside the procedure, each parameter will then be a dummy data structure which will point to each item of the list in turn, in the same way as the parameters of a FOR loop (5.2.1). Conversely, in the statement

```
    PRINT Height,Hres
```

the values of Height and Hres are printed together down the page, and this is possible only if PRINT is able to access both variates simultaneously. In a procedure, this would require the setting PARAMETER=pointer; each parameter is then a pointer, storing the whole list.

You may change some aspects of the Genstat environment within a procedure (5.6.1). This may be the intended purpose of the procedure; but if it is an unwanted side effect, you should reset them afterwards. The RESTORE option allows you to list aspects that would like Genstat to reset automatically when it finishes executing the procedure. Alternatively, you can save and restore most of these aspects explicitly using the directives SET (5.6.1) and GET (5.6.2); however, this is usually less efficient. The exception is the output style, which can be discovered using the ENQUIRE directive (3.3.4) and changed using the OUTPUT directive (3.4.3) as shown in Section 5.4.

The SAVE option allows you to store the contents of the procedure, up to and including ENDPROCEDURE, in a text so that you can edit and redefine it or, for example, print it to a file or save it on backing store. The saved version is a modified form of the original input. Each line of the text contains a single statement; thus, where a statement spans several lines of input, these are concatenated into a single line in the text (deleting the continuation characters). Any line that contains several statements is split. Comments are removed, and any occurrence of several contiguous spaces is replaced by a single space. Also, a colon is placed at the end of each line.

Finally, the WORDLENGTH option allows you to set the wordlength to be used for identifiers, directives, options, parameters and procedures within the procedure. If WORDLENGTH=long, up to 32 characters of each of these names are stored and checked; while if WORDLENGTH=short, no more than eight characters are used. The default is to keep the existing setting of the wordlength (as in the program defining the procedure).

After the PROCEDURE statement, you must define what options and parameters the procedure is to have; this is done by the directives OPTION and PARAMETER respectively. Only one of each of these should be given, and they must appear immediately after the PROCEDURE statement, but it does not matter which of the two you give first. They have very similar syntaxes, except that OPTION has an extra parameter which allows you to indicate whether a list of values or of identifiers is allowed. If you do not wish to define options or parameters for a procedure you can simply omit these directives; alternatively you can use OPTION or PARAMETER but with none of their parameters set, which has precisely the same effect. The OPTION and PARAMETER directives are also used together with the DEFINE directive when extending the Genstat language.

`OPTION` directive

Defines the options of a Genstat procedure with information to allow them to be checked when the procedure is executed.

No options

Parameters

`NAME` = *texts*	Names of the options
`MODE` = *string tokens*	Mode of each option (`e, f, p, t, v,` as for unnamed structures); default `p`
`NVALUES` = *scalars* or *variates*	Specifies allowed numbers of values
`VALUES` = *variates* or *texts*	Defines the allowed values for a structure of type variate or text
`DEFAULT` = *identifiers*	Default values for each option
`SET` = *string tokens*	Indicates whether or not each option must be set (`yes`, `no`); default `no`
`DECLARED` = *string tokens*	Indicates whether or not the setting of each option must have been declared (`yes`, `no`); default `no`
`TYPE` = *texts*	Text for each option, whose values indicate the types allowed (`ASAVE`, `datamatrix` {i.e. pointer to variates of equal lengths as required in multivariate analysis}, `diagonalmatrix`, `dummy`, `expression`, `factor`, `formula`, `LRV`, `matrix`, `pointer`, `RSAVE`, `scalar`, `SSPM`, `symmetricmatrix`, `table`, `text`, `tree`, `TSAVE`, `TSM`, `variate`, `VSAVE`); default `*` meaning no limitation
`COMPATIBLE` = *texts*	Defines aspects to check for compatibility with the first parameter of the directive or procedure (`nvalues, nlevels, nrows, ncolumns, type, levels, labels` {of factors or pointers}, `mode, rows, columns, classification, margins, associatedidentifier, suffixes` {of pointers}, `restriction`)
`PRESENT` = *string tokens*	Indicates whether or not each structure must have values (`yes`, `no`); default `no`
`LIST` = *string tokens*	Whether to allow a list of identifiers (`MODE=p`) or of values (`MODE=v` or `t`) instead of just one (`yes`, `no`); default `no`
`INPUT` = *string token*	Whether the option only supplies input information to the procedure (`yes`, `no`); default `no`

`PARAMETER` directive

Defines the parameters of a Genstat procedure with information to allow them to be checked when the procedure is executed.

No options

Parameters

`NAME` = *texts*	Names of the parameters
`MODE` = *string tokens*	Mode of each parameter (`e, f, p, t, v,` as for

	unnamed structures); default p
NVALUES = *scalars* or *variates*	Specifies allowed numbers of values
VALUES = *variates* or *texts*	Defines the allowed values for a structure of type variate or text
DEFAULT = *identifiers*	Default values for each parameter
SET = *string tokens*	Indicates whether or not each parameter must be set (yes, no); default no
DECLARED = *string tokens*	Indicates whether or not the setting of each parameter must have been declared (yes, no); default no
TYPE = *texts*	Text for each option, whose values indicate the types allowed (ASAVE, datamatrix {i.e. pointer to variates of equal lengths as required in multivariate analysis}, diagonalmatrix, dummy, expression, factor, formula, LRV, matrix, pointer, RSAVE, scalar, SSPM, symmetricmatrix, table, text, tree, TSAVE, TSM, variate, VSAVE); default * meaning no limitation
COMPATIBLE = *texts*	Defines aspects to check for compatibility with the first parameter of the directive or procedure (nvalues, nlevels, nrows, ncolumns, type, levels, labels {of factors or pointers}, mode, rows, columns, classification, margins, associatedidentifier, suffixes {of pointers}, restriction)
PRESENT = *string tokens*	Indicates whether or not each structure must have values (yes, no); default no
INPUT = *string token*	Whether the parameter only supplies input information to the procedure (yes, no); default no

The NAMES parameter of OPTION and PARAMETER defines the names of the options and parameters of the procedure. Each name also defines the identifier of a data structure that will be used, within the procedure itself, to refer to the information transmitted by the relevant option or parameter. When you use the procedure, you have the choice of typing each name in capital letters, or in small letters, or in any mixture of the two; this corresponds to the rules for the names of options and parameters of directives. Within the procedure, however, you need to be more precise, but the exact form of the identifiers will depend upon whether the Genstat environment was set to use short or long "wordlengths" when the procedure was defined. (This is controlled by the WORDLENGTH option of the JOB, SET and PROCEDURE directives.) With long wordlengths, the identifier should be exactly the same as the option name up to the 32nd character; any characters beyond the 32nd are ignored. Alternatively, if short wordlengths have been selected, Genstat forms each identifier by truncating the corresponding option name to no more than eight characters and then converting it into capital letters.

The MODE parameter tells Genstat whether the setting of each option or parameter of the procedure is to be a number (v), or an identifier of a data structure (p), or a string (t), or an expression (e), or a formula (f). These codes are exactly the same as those that indicate the mode of the values to appear within the brackets containing an unnamed structure (1.4.3).

The type of the structure used to represent an option of the procedure depends on the MODE and LIST parameters of the OPTION directive.

For anything other than mode p, the structure will be a dummy. This will point to an expression for mode e, a formula for mode f, and a text for mode t. With mode v, it will point to a scalar if the corresponding setting of the LIST parameter is no, and a variate if LIST=yes.

For mode p and LIST=no, the structure is a dummy, which will point to whichever structure is supplied for the option when the procedure is called; alternatively, when LIST=yes, it is a pointer which will store the list of structures that are supplied. For example, suppose that procedure ALLPOSS which contains the option definitions

```
OPTION \
  NAMES='EXP','FORM','VLN','VLY','TLN','TLY','PLN','PLY'; \
  MODE =  e,     f,     v,     v,     t,     t,     p,     p; \
  LIST = no,    no,    no,    no,   yes,   yes,    no,   yes
```

is called with these options settings:

```
ALLPOSS [EXP=LOG10(X+1); FORM=Variety*Nitrogen; \
  VLN=2; VLY=1,3,5,7; TLN=oneval; TLY=one,two,three; \
  PLN=A; PLY=B,C,D]
```

Inside the procedure it will be as though the identifiers had been defined as follows:

```
DUMMY [VALUE=!E(LOG10(X+1))] EXP
&     [VALUE=!F(Variety*Nitrogen)] FORM
&     [VALUE=2] VLN
&     [VALUE=!(1,3,5,7)] VLY
&     [VALUE='oneval'] TLN
&     [VALUE=!T(one,two,three)] TLY
&     [VALUE=A] PLN
POINTER [VALUE=B,C,D] PLY
```

For parameters, the structures are either all dummies or all pointers, according to the setting of the PARAMETER option of the PROCEDURE directive. If they are pointers, they store all the settings, and the procedure is called only once; if they are dummies, the procedure is called once for every item in the lists. In Example 5.3.2 below, the PARAMETER option is not set, and so it retains the default of dummy. Thus, in line 25, the procedure is called three times; firstly with the dummy PERCENT set to 25 and the dummy RESULT set to Ang25, then with PERCENT set to 50 and RESULT set to Ang50, and finally with PERCENT set to 75 and RESULT set to Ang75. However, if the PROCEDURE statement had been

```
PROCEDURE [PARAMETER=pointer] '%TRANSFORM'
```

PERCENT for example would have been the pointer !P(25,50,75). (Notice that we have called the procedure %TRANSFORM rather than TRANSFORM in order to avoid ambiguity with the directive TRANSFERFUNCTION.)

The other parameters of OPTION and PARAMETER allow the settings that are supplied, when the procedure is called, to be checked automatically.

The NVALUES parameter indicates how many values the structures that are supplied for an option or parameter of mode p may contain. For example,

```
OPTION NAME='X','Y'; NVALUES=3,!(3,4); TYPE='variate'
```

indicates that the variates supplied for X must be of length 3, while those supplied for Y can be of length 3 or 4.

The VALUES parameter can be used with modes t and v to specify an allowed set of values against which those supplied for the option or parameter will be checked. In line 5 of Example 5.3.2, the OPTION statement lists the values that are allowed for METHOD, namely Logit, Comploglog and Angular. The allowed values for mode t define a list of *string tokens* for the option or parameter, that can be used in exactly the same way as the string tokens defined for options or parameters of the ordinary Genstat directives (1.7.3). They can be up to 32 characters in length; characters 33 onwards are ignored. Each value must start with a letter, and may then contain letters or digits. When %TRANSFORM is used, Genstat will check the specified string against those in the VALUES list, using the same abbreviation rules as for string tokens in options and parameters of the ordinary Genstat directives. Thus, for example, to request an angular transformation we need merely put METHOD=A as the first letter A is sufficient to distinguish Angular from Logit and Comploglog. Within %TRANSFORM, Genstat sets METHOD to the full string as defined in the VALUES list, i.e. Angular, and this greatly simplifies its subsequent use (see lines 12, 14 and 16). However, if short wordlengths have been requested, the name is truncated to eight characters and put into capital letters, so Comploglog would become COMPLOGL.

As an example of mode v, this specification would ensure that the numbers supplied for an option NV were all odd integers between one and nine

```
OPTION NAME='NV'; MODE=v; VALUES=!(1,3,5,7,9)
```

The DEFAULT parameter specifies default values to be used if the option or parameter is not set. In Example 5.3.2, METHOD will be set by default to 'Logit'.

The SET parameter indicates whether or not an option or parameter must be set. In the PARAMETER statement in line 8 of Example 5.3.2, we have put SET=yes and so Genstat will check that the parameters of the procedure, PERCENT and RESULT, are both set whenever the procedure is used. The default is SET=no.

The DECLARED parameter specifies whether or not the structures to which options or parameters of mode p are set must already have been declared. For the PERCENT parameter of %TRANSFORM they must have been declared, but for the RESULTS parameter they need not have been. (Any undeclared RESULTS structures will be declared automatically by the CALCULATE statements within the procedure.)

The TYPE parameter can be used to specify a text to indicate the allowed types of the structures to which an option or parameter of mode p is set. The parameters of %TRANSFORM can be either scalars, variates, tables, or any type of matrix (rectangular, symmetric or diagonal). In Example 5.3.2 the COMPATIBLE parameter is then used to specify that the type and number of values of each RESULTS structure must be compatible with those of the equivalent PERCENT structure; this parameter is also available in the OPTION directive, but with both options and parameters, the compatibility checks are against the first parameter of the procedure.

Finally, the PRESENT parameter allows you to indicate that the structure to which an option or parameter is set must have values. The PERCENT parameter must have values, but the RESULTS parameter need not (its values will be calculated within the procedure).

After the OPTION and PARAMETER statements, you then list the statements that are to be executed when the procedure is called: these statements are the sub-program that makes up the procedure. Any data structures defined within the procedure are local to the procedure and cannot be accessed from outside. So you can use any identifiers for the structures, without having to worry about whether they may also be used outside by someone who may later use the procedure. You end these statements making up the procedure by an ENDPROCEDURE statement.

ENDPROCEDURE directive

Indicates the end of the contents of a Genstat procedure.

No options or parameters

Once you have defined a procedure, its subsequent use is very easy. This example shows a procedure to do various transformations of percentages.

Example 5.3.2

```
   2  PROCEDURE '%TRANSFORM'
   3  " Define the arguments of the procedure."
   4    OPTION NAME='METHOD'; MODE=t; \
   5      VALUES=!t(Logit,Comploglog,Angular); \
   6      DEFAULT='Logit'
   7    PARAMETER NAME='PERCENT','RESULT'; \
   8      MODE=p; SET=yes; DECLARED=yes,no; \
   9      TYPE=!t(scalar,variate,matrix,symmetric,diagonal,table);\
        COMPATIBLE=*,!t(type,nvalues); \
        PRESENT=yes,no
      IF METHOD .EQS. 'Logit'
        CALCULATE RESULT = LOG( PERCENT / (100-PERCENT) )
      ELSIF METHOD .EQS. 'Comploglog'
        CALCULATE RESULT = LOG( -LOG((100-PERCENT)/100) )
      ELSIF METHOD .EQS. 'Angular'
        CALCULATE RESULT = ANGULAR(PERCENT)
      ENDIF
    ENDPROCEDURE

    VARIATE    [VALUES=10,20...90] Every10%
    " default setting 'logit' for METHOD "
    %TRANSFORM Every10%; RESULT=Logit10%
    PRINT      Every10%,Logit10%; DECIMALS=0,3

  Every10%    Logit10%
        10      -2.197
        20      -1.386
        30      -0.847
        40      -0.405
        50       0.000
        60       0.405
        70       0.847
        80       1.386
        90       2.197

    %TRANSFORM [METHOD=A] 25,50,75; RESULT=Ang25,Ang50,Ang75
    PRINT      Ang25,Ang50,Ang75

     Ang25       Ang50       Ang75
     30.00       45.00       60.00
```

When you define a procedure, Genstat usually checks that any procedures that it calls are available in the program or in an attached procedure library. However, this can create problems if you have procedures that call each other. For example, Genstat is happy to execute programs where a procedure, `A` say, calls other procedures that themselves call procedure `A`, but it can then be difficult to work out an order in which to define the procedures successfully.

The original solution was to set up a library of dummy procedures (with option and parameter definitions but no executable statements) to attach to Genstat while the real procedures were defined. In Release 10, however, a better solution is provided by the `CALLS` directive. If you specify a `CALLS` statement in a procedure, listing the procedure that it calls, Genstat will regard these as a set of "trusted" sub-procedures, and assume that they will become available before the procedure is executed. (If not, you will get a fault diagnostic then!) You can thus define the procedures in any convenient order.

`CALLS` directive

Lists library procedures called by a procedure.

No options

Parameter

identifiers	Names of the called procedures

The `CALLS` statement must come immediately after the option and parameter definitions (using the `OPTION` and `PARAMETER` directives), and before any executable statements. It has a single parameter, that lists the names of the procedures that are called.

5.3.3 Forming and using your own procedure libraries

A *procedure library* is a particular kind of backing-store file that is used to store procedures. It can be used like any other backing store file: you can store procedures in the file, then retrieve them later for further use, using the methods described in 3.5. However you will usually find a library more convenient to use when it is attached to one of the input channels reserved just for procedure libraries. You can then only read procedures from the file and you cannot add new procedures; but the procedures are retrieved from the library automatically, as described at the start of this section.

Several libraries can be attached to a Genstat job. The standard Genstat procedure library is attached automatically, and you may have local site libraries that are also attached automatically. In Genstat *for Windows*, you can arrange this to happen by putting the libraries into the system add-in folder, and then using the Procedure Libraries menu (opened by selecting the Attach sub-option of the Procedure Libraries option of the Tools menu on the menu bar); see the on-line help for details. There is also a user add-in folder that can to separate an individual user's libraries from those that are distributed across a site.

You can also attach libraries explicitly, using the `OPEN` directive (3.3.1). For example:

```
OPEN 'Graphlib.glb'; CHANNEL=2; FILETYPE=procedurelibrary
```

Maintaining a procedure library is more efficient if the procedures are stored in separate subfiles, and accessing is more efficient if you give the subfiles the same names as the procedures. To store procedures you use the `STORE` directive (3.5.3), for example:

```
STORE [CHANNEL=1; SUBFILE=Jacknife; PROCEDURE=yes] Jacknife
```

Some procedures may contain references to auxiliary procedures for performing particular parts of an analysis; in this case the searching of the library is more efficient if the additional procedures are contained in the same subfile as the main procedure: for example

```
STORE [CHANNEL=1; SUBFILE=Plot; PROCEDURE=yes] \
  Plot,Scalex,Scaley
```

While you are developing a procedure library you will need to use it like any other backing-store file, retrieving any procedures that are required by using `RETRIEVE`. To edit a procedure library you can use either of the directives `STORE` (3.5.3) or `MERGE` (3.5.6). You can display the contents of a library and subfiles using `CATALOGUE` (3.5.5).

Help information for user procedure libraries can be supplied in Genstat *for Windows* by putting a Windows (`.chm`) help file alongside the procedure library (`.glb`) file in the add-in folder. The Windows Help menu is then extended to contain a sub-option with the name of the procedure library, in its User Libraries option. Clicking on that sub-option opens the file at its contents page. The names of the procedures can be added to the context-sensitive help by including a topic in the help file for each procedure, and naming these by the procedure names (in full). Users can then access the help for a procedure by placing the curser in the name of the procedure in Genstat *for Windows*' Output window, and pressing the `F1` key (in the same way as for the standard Genstat commands, functions and terminology).

5.4 Useful commands for procedure writers

You should use the `CAPTION` directive (3.2.3) to put titles into the output from your procedure. These will then be compatible with the titles in output from the ordinary Genstat commands, such as `ANOVA` or `FIT`. Also, they will be customized automatically to suit the current output style. For example, in the plain-text style they will be underlined by lines of equals or minus characters, whereas in formatted styles they will be printed in larger, coloured or bold fonts. Output can be constructed using the `PRINT` directive (3.2.1). You should use the `HEADING` parameter of `PRINT` to put headings above columns of numbers (from variates of factors) or strings (from texts), rather than 2 separate `PRINT` statements: for example

```
PRINT Y,Fitted,Residual; FIELD=18,14,10;\
  HEADING='Response variate','Fitted value','Residual'
```

instead of

```
PRINT [IPRINT=*; SQUASH=yes] \
  HEADING='Response variate','Fitted value','Residual';\
  FIELD=18,14,10;
& Y,Fitted,Residual; FIELD=18,14,10
```

By putting all the output into a single PRINT allows Genstat to line all the information up in a single table if the output is in a formatted style. If you do need to construct output using several separate PRINT statements, you may need to switch the output temporarily to plain text, by the statement

```
OUTPUT [STYLE=plaintext]
```

The original style can be restored automatically when the procedure ends, by including outstyle amongst the settings of the RESTORE option in the PROCEDURE statement. Alternatively, you can discover the current style by putting

```
SCALAR  Chan
ENQUIRE Chan; FILETYPE=output; OUTSTYLE=Style
```

Chan is automatically set to the number of the current output channel, and Style is formed into a text containing either 'formatted' or 'plaintext'. The original style can then be restored automatically by

```
OUTPUT [STYLE=#Style] Chan
```

In the plain-text output style columns of output are lined up using space characters, while in the formatted styles they are defined using special codes. If, however, you want to use PRINT to output a "sentence" of information, you may want the columns separated by spaces even when the style is not plain text. You should then set option STYLE=plaintext. For example

```
PRINT [STYLE=plain] 'There are',Df,'degrees of freedom.';\
  FIELD=9,2,20; DECIMALS=0
```

The GET directive (5.6.2) allows you to obtain details about the current state of the Genstat environment or the settings of special structures like the model formula most recently specified by the TREATMENTSTRUCTURE directive, and the SET directive (5.6.1) allows you to modify any of these. In particular, you can use the CAPTIONS option of SET to suppress unwanted captions from any of the commands that the procedure uses. However, unless the changes are part of the intended purpose of the procedure, they should be reset at the end of the procedure. This can be done by using GET to store the information, and then SET to reset it; alternatively you can use the RESTORE option of the PROCEDURE directive (5.3.2).

There are several functions that you may find useful when writing procedures. You might use these either in CALCULATE (4.1.1), or in the program-control directives (5.2). Some of the functions enable you to access information about the structures that have been supplied in the options or parameters of the procedure. For example: the function NVALUES allows you to find out the length of a structure, NROWS enables you to find out the number of rows of a matrix, and so on (4.2.2). Alternatively you can use the GETATTRIBUTE directive (2.11.3). You might want to use this information to check that the supplied structures are suitable for the operations that the procedure is to carry out; or you might use it in the definition of the local structures required within the procedure.

You can use the SET function (4.2.6), or its converse UNSET, to check whether the user has set a particular option or parameter. If this option or parameter is necessary for some particular section of the procedure to be executed you might want to use a block-if structure (5.2.2), or you might use the EXIT directive to leave the procedure altogether. The ASSIGN directive (4.9.1) provides a convenient way of setting the dummies to some default structure within the procedure (or even to structures outside the procedure). For example, the following statements assign

PERCENT (if unset) to one of two different variates, according to whether this is a batch or an interactive run; then RESULT is assigned, if necessary, to Res.

```
IF UNSET(PERCENT)
  GET [ENVIRONMENT=Env]
  IF Env['run'].eqs.'batch'
    ASSIGN !(1,2.5,5,7.5,(10,15...90),92.5,95,97.5,99);\
      POINTER=PERCENT
  ELSE
    ASSIGN !(1,5,10,25,50,75,90,95,99); POINTER=PERCENT
EXIT [CONTROLSTRUCTURE=procedure]
ENDIF
ASSIGN [METHOD=preserve] RESULT; Res
```

The setting `METHOD=preserve` in `ASSIGN` will preserve any existing setting of `RESULT` but assign it to `Res` if it is unset.

Chapter 4 describes many other useful commands. In addition to general numerical calculations (4.1 and 4.2), it also covers manipulation of data structures (4.4) (4.4, 4.5 and 4.6), Boolean arithmetic and set operations (4.3), text manipulation (4.7), model formulae (4.8), matrix calculations (4.10) and calculations and manipulation of tables (4.11).

You can use other procedures from within a procedure; in fact you can even call the procedure itself, so you can write recursive programs. However, these auxiliary procedures must either be declared within your procedure using the `CALLS` directive (5,3,2), or they must be available within your program when the procedure is defined: i.e. they must either have been defined earlier within your program or be available within one of the libraries attached to your job. You cannot define a procedure within another procedure or within any other control structure. The `Utility` module of the procedure library contains several procedures useful to procedure writers, including `CHECKARGUMENT` which can be used to check various aspects of option and parameter settings.

You are allowed to redefine an existing procedure if you wish to change any of the statements that it contains. To do this you specify the `PROCEDURE` statement, as usual, followed by the statements making up the new version of the procedure, and then an `ENDPROCEDURE` statement. However, you are not allowed to change the option or parameter definitions, and if there are any changes in the `OPTION` or `PARAMETER` statements, Genstat will give an error diagnostic.

If you are running short of workspace, remember that you can use the `DELETE` directive (2.10.1) to delete any procedures that are no longer required, or which can be accessed again from a library if they should be needed. For example

```
DELETE [PROCEDURE=yes] %TRANSFORM
```

to delete procedure `%TRANSFORM` or

```
DELETE [PROCEDURE=yes; LIST=all]
```

to delete all the procedures that are currently in store.

There are also several specialised directives, with names prefixed by `%`, that may be useful.

`%LOG`	adds text into the Input Log window in the Genstat client
`%MESSAGEBOX`	displays text in a dialog in the Genstat client
`%OPEN`	open a binary file for use with `%WRITE`
`%FPOSITION`	returns the current position in the binary file opened by `%OPEN`
`%WRITE`	writes values of data structures to a binary file opened by `%OPEN`
`%CLOSE`	closes the binary file opened by `%OPEN`
`%SLEEP`	pauses execution of the server for a time specified in seconds
`%TEMPFILE`	creates a unique temporary file in the Genstat temporary folder

Details are in the *Genstat Reference Manual, Part 2 Directives*.

5.4.1 Diagnostics

The FAULT directive is the recommended way of generating diagnostics (i.e. faults, warnings or messages) from a procedure.

FAULT directive

Checks whether to issue a diagnostic, i.e. a fault, warning or message.

Options

DIAGNOSTIC = *string token*	Severity of the diagnostic (fault, warning, message); default faul
FAULT = *text*	Diagnostic code; default 'UF 1' for fault, 'UF 2' for warning
EXPLANATION = *text*	Explanatory information
NCALLS = *scalar*	Number of calls from the main procedure (whose name should be used in fault or warning messages); default 0

Parameter

expression	Logical expression to test whether or not to give the diagnostic

The diagnostics are printed in the standard Genstat format. So, for example, faults and warnings are recognized by Genstat *for Windows*, and added to the Event Log. Also, the diagnostic will be suppressed (like those from Genstat directives) if any user of the procedure has requested that by using the DIAGNOSTICS option of the SET directive (5.6.1).

There is a single parameter, which supplies a logical expression to decide whether or not to give the diagnostic; if this is omitted, the diagnostic is always given. The FAULT option defines the code to identify a fault or warning; this has a default of 'UF 1' for a fault and 'UF 2' for a warning. (Messages always begin with the standard prefix "Message: ".) The EXPLANATION option allows you to supply some explanatory information.

For example, in a regression procedure, you might put

```
FAULT [DIAGNOSTIC=fault; FAULT='VA 6';\
  EXPLANATION='Y-variate must contain at least 2 values']\
  NOBSERVATIONS(Y) < 2
```

Then, if the y-variate has less than two non-missing values, Genstat will give a "VA 6" fault, and execution of the procedure will stop.

The NCALLS option is useful if you want to give diagnostics from a subsidiary procedure, called by the main procedure. If this was called directly by the main procedure, you can set NCALLS=1 to ensure that a fault or warning is identified as having come from the main, rather than from the subsidiary, procedure.

You may want to perform some of your own error checking within the procedure, instead of allowing the directives or procedures that it calls to give diagnostics. The DIAGNOSTIC option of the SET directive (5.6.1) allows you to suppress various classes of diagnostic. You can use the GET directive (5.6.2) to access the current value of Genstat's internal fault indicator so that you can ascertain which diagnostic (if any) occurred most recently (and you can also use the FAULT option of SET to clear the last diagnostic to avoid any confusion about when the diagnostic occurred). If you decide that you do want to print the diagnostic, you can use the DISPLAY directive. In fact you can use the FAULT option of DISPLAY to print any Genstat diagnostic; if FAULT is not set, DISPLAY prints the most recent diagnostic.

DISPLAY directive

Prints, or reprints, diagnostic messages.

Options

PRINT = *string token*	What information to print (diagnostic); default diag
CHANNEL = *identifier*	Channel number of file, or identifier of a text to store output; default current output file
FAULT = *text*	Specifies the fault message to print (for example, FAULT='VA 4' prints the message "Values not set"); default is to print the last diagnostic message

No parameters

5.4.2 Private data structures: the WORKSPACE directive

If you are writing a suite of procedures to provide an integrated set of facilities, you might want to pass private information between them, unseen by the user. (For example, the regression directives FIT, ADD, DROP etc automatically pass information about the current status of the model that is being fitted.) This can be done using the WORKSPACE directive.

WORKSPACE directive

Accesses private data structures for use in procedures.

No options

Parameters

NAME = *texts*	Texts, each containing a single line, to give the names used to identify the private data structures
DUMMY = *identifiers*	Dummy structure to be used to refer to each private data structure

The WORKSPACE directive is intended particularly for writers of procedures. It allows data to be accessed within a number of procedures, and in the main program if needed. You merely need to decide how to label your workspace "areas". Genstat reserves a data structure for each one, and WORKSPACE allows you to link this to a dummy (of your choice) within any procedure or in the outer program itself. For example

```
WORKSPACE 'AUNBALANCED work'; Wspace
TEXT     [VALUES=Yvar,Factopt] Wlabels
POINTER  [NVALUES=Wlabels] Wspace
VARIATE  Wspace['Yvar']
SCALAR   Wspace['Factopt']
```

names the area 'AUNBALANCED work' and sets the dummy Wspace to the associated data structure. The data structure is then defined to be a pointer with two values, the variate Wspace['Yvar'] and the scalar Wspace['Factopt']. A similar WORKSPACE statement can then be used later on (in another procedure) to access the same information. For example

```
WORKSPACE 'AUNBALANCED work'; Abwork
```

links the dummy Abwork to the pointer, allowing us to refer to Abwork['Yvar'] and Abwork['Factopt']. This will be used particularly within the procedure library, to link suites of associated procedures so, for safety, you should avoid prefixing the name of any workspace of your own by G5PL.

5.4.3 Execution of macros

There are two ways in which you can insert the contents of a text as a macro into the statements within a procedure. With the `##` operator, the contents are inserted at the time that the procedure is defined. For example

```
TEXT [VALUES='  CALCULATE V = VARIANCE(X)',\
  '  IF V>0',\
  '    CALCULATE X = (X - MEAN(X))/V',\
  '  ELSE',\
  '    CALCULATE X = CONSTANT(''missing'')',\
  '  ENDIF'] Calcs
PROCEDURE 'STANDARD'
PARAMETER NAME='X'
##Calcs
ENDPROCEDURE
```

will define the procedure `STANDARD` as

```
PROCEDURE 'STANDARD'
PARAMETER NAME='X'
  CALCULATE V = VARIANCE(X)
  IF V>0
    CALCULATE X = (X - MEAN(X))/V
  ELSE
    CALCULATE X = CONSTANT('missing')
  ENDIF
ENDPROCEDURE
```

(Notice that the quotes around `missing` need to be given twice to tell Genstat that these are a part of the string and are not intended to mark the end of the string; see 1.4.2.)

Alternatively, you may want to take the contents of the text and execute them only at the same time as the procedure is executed. This facility is provided by the `EXECUTE` directive.

`EXECUTE` directive

Executes the statements contained within a text.

No options

Parameter

texts	Statements to be executed

Example 5.4.3 shows a rather simple use of `EXECUTE`, to execute different statements on each pass through a loop.

Example 5.4.3

```
TEXT [VALUES='SCALAR X; VALUE=12'] T1
&    [VALUES='DELETE [REDEFINE=yes] X','TEXT [VALUE=Twelve] X'] T2
FOR T=T1,T2
  EXECUTE T
  PRINT X
ENDFOR

        X
    12.00


        X
   Twelve
```

5.4.4 Incrementing a multi-digit counter

The `COUNTER` directive is useful if you want to increment a counter made up of several digits that recycle to limits that may be different from ten. For example, times in seconds, minutes and hours, or measurements in inches, feet and yards.

`COUNTER` directive

Increments a multi-digit counter using non base-10 arithmetic.

Options

`NREQUIRED` = *scalar*	Specifies the number of values required for the counter; default 2
`NFOUND` = *scalar*	Saves the number of counter values that could be formed
`DIRECTION` = *string token*	Specifies the direction of the sequence of increments to the counter (`ascending`, `descending`); default `asce`

Parameters

`START` = *scalars*	Provides the starting values for the digits in the counter
`END` = *scalars*	Can provide values to define the end of the sequence of counter values
`STEP` = *scalars*	Specifies the amount by which to increment each digit of the counter
`BASE` = *scalars*	Specifies the base of the numbers used for each digit
`DIGITSEQUENCE` = *variates*	Saves the sequence of values generated for each digit

The parameters provide details of the digits in the counter, all in scalars. The `BASE` parameter specifies the base of the numbers used for each digit (e.g. 60 for seconds and minutes, and 24 for hours). The `START` parameter supplies the starting values of the digits, ranging from zero to `BASE` minus one. The `STEP` parameter specifies the size of the increment for each digit. The digits are updated from the right-hand side and, when one goes beyond its limit, the next one is incremented by an extra value of one for an ascending sequence, or minus one for a descending sequence. The `DIGITSEQUENCE` saves the sequence of values formed for each digit of the counter, in variates.

The `END` parameter can specify values to define the end of the sequence. If a value is specified for every digit, the sequence ends when the next set of digits would go beyond those supplied by `END`: above `END` for an ascending sequence, or below for a descending sequence. (See the `DIRECTION` option.) Otherwise, the sequence ends when all the digits would go beyond their limits: `BASE` minus one for an ascending sequence, or zero for a descending sequence.

The `NREQUIRED` option specifies the number of values that are required for the counter. The default is 2, i.e. `START` and one other. The `NFOUND` option can save the number of values that have been formed. The `DIRECTION` option controls whether the sequence of counter values should be regarded as ascending or descending, when checking for the end of the sequence. The default is ascending.

Example 5.4.4 counts in inches from 1 foot 11 inches to 2 yards. (There are 12 inches in a foot, and 3 feet in a yard.)

Example 5.4.4

```
" Count in inches from 1 foot 11 inches to 2 yards "
COUNTER [NREQUIRED=99] 1,0,11; END=2,0,0; STEP=0,0,1;\
        BASE=1760,3,12; DIGITSEQUENCE=yard,foot,inch
PRINT   yard,foot,inch; DECIMALS=0
```

```
        yard        foot        inch
           1           0          11
           1           1           0
           1           1           1
           1           1           2
           1           1           3
           1           1           4
           1           1           5
           1           1           6
           1           1           7
           1           1           8
           1           1           9
           1           1          10
           1           1          11
           1           2           0
           1           2           1
           1           2           2
           1           2           3
           1           2           4
           1           2           5
           1           2           6
           1           2           7
           1           2           8
           1           2           9
           1           2          10
           1           2          11
           2           0           0
```

5.4.5 Information about commands

The COMMANDINFORMATION directive enables you to discover whether a command is present in your version of Genstat and, if so, whether it is a directive or a procedure.

COMMANDINFORMATION directive

Provides information about whether (and how) a command has been implemented.

No options

Parameters

NAME = *texts*	Single-line texts supplying the names of the commands
IMPLEMENTATION = *texts*	Single-line texts set to 'directive', 'procedure' or a null string ('') according to the type of command
CHANNEL = *scalars*	Saves the channel for a procedure from a procedure library
PRESENTNOW = *scalars*	Logical set to one if the command is now present, or zero otherwise

The name of the command must be supplied in a single-value text, using the NAME parameter. The IMPLEMENTATION parameter can save another single-valued text, which is set to 'directive' or 'procedure' according to the type of command. If the command is not present, it is set to a null string ('') .

The PRESENTNOW parameter provides another, possibly simpler, way of discovering whether the directive or procedure is currently present within Genstat. This saves a scalar containing the value one if the command is present, or zero otherwise.

For procedures accessed from a procedure library, the CHANNEL option can save a scalar with the number of the channel to which the library is attached. This contains a missing value if the command is not present as a procedure. It contains zero if the procedure was created in this job (using the PROCEDURE directive). The channel number for the official procedure library is 12, and the channel for the local procedure library is 11.

Example 5.4.5 continues Example 5.3.2, to show the information obtained about the procedure `%TRANSFORM` defined there, as well as that for a directive (`CAPTION`), a library procedure (`DOTPLOT`), and a non-existent command (`NOTONE`).

Example 5.4.5

```
  28  COMMANDINFORMATION '%TRANSFORM','CAPTION','DOTPLOT','NOTONE';\
  29    IMPLEMENTATION=tranimp,capimp,dotimp,notimp;\
  30    CHANNEL=tranchan,capchan,dotchan,notchan;\
  31    PRESENT=trancheck,capcheck,dotcheck,notcheck
  32  PRINT tranimp,tranchan,trancheck

  tranimp    tranchan   trancheck
procedure           0       1.000

  33  &     capimp,capchan,capcheck

   capimp     capchan    capcheck
directive           *       1.000

  34  &     dotimp,dotchan,dotcheck

   dotimp     dotchan    dotcheck
procedure       12.00       1.000

  35  &     notimp,notchan,notcheck

notimp     notchan    notcheck
                 *           0
```

5.4.6 Information about syntax

The `SYNTAX` directive enables you to obtain details of the syntax of a command (i.e. a directive or a procedure) and the source code of a procedure.

SYNTAX directive

Obtains details of the syntax of a command and the source code of a procedure.

No options

Parameters

`COMMAND` = *texts*	Single-line texts specifying the commands
`NOPTIONS` = *scalars*	Number of options for each command
`NPARAMETERS` = *scalars*	Number of parameters for each command
`NAME` = *texts*	Names of the options, and then the parameters, of each command
`MODE` = *texts*	Modes of the options and parameters
`NVALUES` = *pointers*	Number of values allowed for the options and parameters
`VALUES` = *pointers*	Allowed values for the options and parameters
`DEFAULT` = *pointers*	Default values for the options and parameters
`SET` = *texts*	Whether the options and parameters must be set
`DECLARED` = *texts*	Whether the options and parameters must have been declared
`TYPE` = *pointers*	Allowed types for the options and parameters
`COMPATIBLE` = *pointers*	Aspects of the options and parameters that must be compatible with the first parameter

PRESENT = *texts*	Whether the options and parameters must have values
LIST = *texts*	Whether the options have more than one setting (not relevant for the parameters
INPUT = *texts*	Whether the options and parameters only supply input information
DEFINITION = *texts*	Saves statements to define the syntax
SOURCE = *texts*	Saves the source code of a procedure

The name of the command must be supplied in a single-value text, using the COMMAND parameter. The NOPTIONS parameter gives its number of options, and the NPARAMETERS parameter gives the number of parameters.

The other parameters give details of the options and parameters. These correspond to the parameters of the OPTION and PARAMETER directives.

The NAMES parameter saves a text containing the names of the options (if any), followed by the names of any parameters.

The MODE parameter saves a text giving the modes of the options and parameters: whether their settings should be a number (v), or an identifier of a data structure (p), or a string (t), or an expression (e), or a formula (f). These codes are exactly the same as those that indicate the mode of the values to appear within the brackets containing an unnamed structure.

The NVALUES saves a pointer defining how many values the structures that are supplied for options and parameters of mode p may contain. The element of the pointer is a scalar there is only one possibility, and a variate if there are several.

The VALUES saves a pointer containing the allowed set of values that may have been defined for options and parameters with modes t and v. The element of the pointer will be a text for an option or parameter of mode t, and either a scalar or a variate for an option or parameter of mode v.

The DEFAULT parameter saves a pointer containing the default settings that may have been defined for the options and parameters with modes t and v.

The SET parameter saves a text containing 'yes' or 'no' according to whether or not the options and parameters must be set.

The DECLARED parameter saves a text containing 'yes' or 'no' according to whether or not the options and parameters of mode p must be set to a data structure that has already been declared.

The TYPE parameter saves a pointer containing a text to indicate the allowed types of the structures to which each option and parameter of mode p can be set.

The COMPATIBLE parameter saves a pointer containing a texts to specify aspects of the options and parameters that must be compatible with the first parameter.

The PRESENT parameter saves a text containing 'yes' or 'no' according to whether or not the options and parameters must be set to a data structure that has values.

The INPUT parameter saves a text containing 'yes' or 'no' according to whether or not the options and parameters are be used only to provide input to the command.

The DEFINITION parameter can save statements, in a text, to define the syntax. These start with a DEFINE statement for a directive or a PROCEDURE statement for a procedure, then an OPTION statement to define any options, and a PARAMETER statement to define any parameters.

The SOURCE parameter can save the source code of a procedure. This can be useful if you have a library containing the procedure, but no longer have the original source file. Note, though, that the source that you save will not be identical to the original source. When procedures are defined within Genstat, their source code is processed to remove comments and extraneous spaces in order to save storage space (as shown when the source of the procedure %TRANSFORM is printed at the end of Example 5.4.6 below). It also inserts colons to end the statements explicitly.

Example 5.4.6 continues Example 5.3.2 ,to show how to obtain details of the syntax, definition and source code of the procedure `%TRANSFORM` defined at the start of the example. The `FOR` loop in lines 45-61 assigns default texts to the pointer elements that have not needed to be defined by the `SYNTAX` directive to make the subsequent printing clearer.

Example 5.4.6

```
  37  SYNTAX '%TRANSFORM'; NOPTIONS=nopt; NPARAMETERS=npar; NAME=names;\
  38         MODE=modes; NVALUES=nvals; VALUES=values; DEFAULT=default;\
  39         SET=set; DECLARED=declared; TYPE=types; COMPATIBLE=compat;\
  40         PRESENT=present; LIST=list; INPUT=input;\
  41         DEFINITION=define; SOURCE=source
  42  PRINT  nopt,npar; DECIMALS=0


        nopt        npar
           1           2

  43  &      names,modes,set,declared,list,input,present;\
  44         FIELD=*,6,4,9,5,6,8; JUST=left,6(right)

names   modes set declared list input present
METHOD      t  no       no   no    no      no
PERCENT     p yes      yes         no     yes
RESULT      p yes       no         no      no

  45  FOR [INDEX=i; NTIMES=nopt+npar]
  46    IF NMV(NVALUES(nvals[i]))
  47      ASSIGN 'No nvals'; POINT=nvals; ELEMENT=i
  48    ENDIF
  49    IF NMV(NVALUES(values[i]))
  50      ASSIGN 'No values'; POINT=values; ELEMENT=i
  51    ENDIF
  52    IF NMV(NVALUES(default[i]))
  53      ASSIGN 'No default'; POINT=default; ELEMENT=i
  54    ENDIF
  55    IF NMV(NVALUES(types[i]))
  56      ASSIGN 'No types'; POINT=types; ELEMENT=i
  57    ENDIF
  58    IF NMV(NVALUES(compat[i]))
  59      ASSIGN 'No compat'; POINT=compat; ELEMENT=i
  60    ENDIF
  61  ENDFOR
  62  PRINT  [SQUASH=yes; ORIENT=across] nvals[]
 nvals['METHOD'] No nvals
nvals['PERCENT'] No nvals
 nvals['RESULT'] No nvals
  63  &      values[]
values['METHOD']      LOGIT COMPLOGLOG    ANGULAR
values['PERCENT'] No values
values['RESULT'] No values
  64  &      default[]
 default['METHOD']      LOGIT
default['PERCENT'] No default
 default['RESULT'] No default
  65  &      types[]
types['METHOD'] No types
types['PERCENT']          scalar         variate          matrix symmetricmatrix
types['PERCENT']  diagonalmatrix           table
types['RESULT']          scalar         variate          matrix symmetricmatrix
types['RESULT']  diagonalmatrix           table
  66  &      compat[]
compat['METHOD'] No compat
compat['PERCENT'] No compat
compat['RESULT']    type nvalues
  67  PRINT  define;  JUST=left & source; JUST=left

define
PROCEDURE [RESTORE=; WORDLENGTH=long] '%TRANSFORM'
OPTIONS \
  NAME='METHOD';\
  MODE='t';\
  VALUES=!t(LOGIT,COMPLOGLOG,ANGULAR);\
  DEFAULT=!t(LOGIT);\
  SET='no';\
  DECLARED='no';\
  PRESENT='no';\
  LIST='no';\
  INPUT='no'
PARAMETERS \
  NAME='PERCENT','RESULT';\
  MODE='p','p';\
  SET='yes','yes';\
  DECLARED='yes','no';\
  TYPE=!t(scalar,variate,matrix,symmetricmatrix,diagonalmatrix,table),!t(\
    scalar,variate,matrix,symmetricmatrix,diagonalmatrix,table);\
  COMPATIBLE=*,!t(type,nvalues);\
  PRESENT='yes','no';\
  INPUT='no','no'


source
 IF METHOD .EQS. 'Logit':
 CALCULATE RESULT = LOG( PERCENT / (100-PERCENT) ):
 ELSIF METHOD .EQS. 'Comploglog':
 CALCULATE RESULT = LOG( -LOG((100-PERCENT)/100) ):
 ELSIF METHOD .EQS. 'Angular':
 CALCULATE RESULT = ANGULAR(PERCENT):
 ENDIF:
ENDPROCEDURE:
```

5.5 Debugging Genstat programs

If you are writing a general program in the Genstat language (as in any other high-level language) you may often find that your program is syntactically correct and can be executed by Genstat, but nevertheless produces the wrong answers: somewhere in the logic of your program you have made a mistake. To allow such errors to be identified and corrected, Genstat has two directives, BREAK and DEBUG, that allow you to interrupt the execution of your program. You can then execute other statements, for example to examine the contents of data structures or modify their values, or even to exit from a control structure. This is particularly useful inside a procedure: the data structures used by the procedure are local and cannot normally be accessed from outside; during a break you remain within the procedure and so all the local data structures can be accessed. The BREAK directive allows you to insert breakpoints explicitly; so you must plan its use in advance when you are writing the code. Alternatively you can use DEBUG to insert breakpoints implicitly. This allows you for example to debug an existing procedure without having to edit and redefine it.

5.5.1 Breaking into the execution of a program

BREAK directive

Suspends execution of the statements in the current channel or control structure and takes subsequent statements from the channel specified.

Option

CHANNEL = *scalar*	Channel number; default 1

Parameter

expression	Logical expression controlling whether or not the break takes place

The BREAK directive allows you to halt the execution of the current set of statements temporarily so that you can execute some other statements. If the parameter is not set, the break will always take place. Alternatively, you can specify a logical expression and then the break will take place only if this produces a *true* (i.e. non-zero and non-missing) result.

The CHANNEL option determines where the statements to be executed during the break are to be found. Usually (and by default) they are in channel 1. The statements are read and executed, one at a time, until an ENDBREAK statement is reached, at which point control returns to the statements originally being executed.

ENDBREAK directive

Returns to the original channel or control structure and continues execution.

No options or parameters

BREAK provides a convenient way of interrupting a loop or a procedure so that you can read one set of output before the next is produced, as shown in Example 5.5.1.

Example 5.5.1

```
   2  VARIATE [NVALUES=13] X,Y,LogY
   3  READ X,Y

    Identifier   Minimum      Mean   Maximum    Values   Missing
             X     6.000     30.00     60.00        13         0
             Y     72.50     95.42     115.9        13         0

  17  CALCULATE LogY = LOG(Y)
  18  FOR Dum=Y,LogY
  19     MODEL Dum
  20     TERMS X
  21     FIT [PRINT=summary] X
  22     BREAK
  23     RDISPLAY [PRINT=estimates]
  24     BREAK
  25  ENDFOR

25..........................................................................

Regression analysis
===================

Summary of analysis
-------------------

Source        d.f.         s.s.         m.s.      v.r.
Regression       1       1831.9      1831.90     22.80
Residual        11        883.9        80.35
Total           12       2715.8       226.31

Percentage variance accounted for 64.5
Standard error of observations is estimated to be 8.96.

* MESSAGE: the following units have large standardized residuals.
         Unit     Response    Residual
           10       115.90        2.04

* MESSAGE: the following units have high leverage.
         Unit     Response    Leverage
            1        78.50        0.34

***** break at statement 5 in for loop
" RDISPLAY [PRINT=estimates]"
  26  ENDBREAK

26..........................................................................
```

```
Regression analysis
===================

Estimates of parameters
-----------------------

Parameter      estimate         s.e.     t(11)
Constant         117.57         5.26     22.34
X                -0.738        0.155     -4.77

***** break at statement 7 in for loop
"ENDFOR"
  27  ENDBREAK

27.........................................................................

Regression analysis
===================

Summary of analysis
-------------------

Source        d.f.         s.s.         m.s.      v.r.
Regression       1      0.21732     0.217322     24.07
Residual        11      0.09931     0.009028
Total           12      0.31663     0.026386

Percentage variance accounted for 65.8
Standard error of observations is estimated to be 0.0950.

* MESSAGE: the following units have high leverage.
         Unit     Response    Leverage
            1       4.3631        0.34

***** break at statement 5 in for loop
" RDISPLAY [PRINT=estimates]"
  28  ENDBREAK

28.........................................................................

Regression analysis
===================

Estimates of parameters
-----------------------

Parameter      estimate         s.e.     t(11)
Constant         4.7876       0.0558     85.83
X              -0.00804      0.00164     -4.91

***** break at statement 7 in for loop
"ENDFOR"
  29  ENDBREAK
```

5.5.2 Putting automatic breaks into a program

`DEBUG` directive

Puts an implicit `BREAK` statement after the current statement and after every `NSTATEMENTS` subsequent statements, until an `ENDDEBUG` is reached.

Options

`CHANNEL` = *scalar*	Channel number; default 1
`NSTATEMENTS` = *scalar*	Number of statements between breaks; default 1
`FAULT` = *string token*	Whether to invoke `DEBUG` only at the next fault (`yes`, `no`); default `no`

No parameters

`ENDDEBUG` directive

Cancels a `DEBUG` statement.

No options or parameters

The straightforward use of `DEBUG` causes an immediate break, and then further breaks at regular intervals until you issue an `ENDDEBUG` statement. Alternatively, by setting option `FAULT=yes`, you can arrange for Genstat to continue until the next fault diagnostic, and then break.

The interval before each further break is specified by the `NSTATEMENTS` option; by default, breaks take place after every statement.

During the breaks, Genstat takes statements from the channel specified by the `CHANNEL` option; by default they are taken from channel 1.

Each individual break is terminated by an `ENDBREAK`, exactly like a break invoked explicitly by the `BREAK` directive (5.5.1).

For example:

Example 5.5.2

```
   2  PROCEDURE 'POLAR'
   3     PARAMETER 'X','Y','R','THETA'
   4     " Takes (x,y) and returns (r,theta) "
   5     CALCULATE R = SQRT(X*X + Y*Y)
   6     CALCULATE THETA = ARCCOS(X/R)
   7     CALCULATE THETA = THETA + 2*(3.14159 - THETA)*(Y < 0)
   8  ENDPROCEDURE
   9  SCALAR Xpos,Ypos; VALUE=3,4
  10  DEBUG
  11  POLAR Xpos; Y=Ypos; R=Radius; THETA=Angle
***** break at statement 1 in procedure POLAR
"  CALCULATE R = SQRT(X*X + Y*Y)"
  12  ENDBREAK
***** break at statement 2 in procedure POLAR
" CALCULATE THETA = ARCCOS(X/R)"
  13  PRINT R

      Radius
       5.000

  14  ENDBREAK
***** break at statement 3 in procedure POLAR
" CALCULATE THETA = THETA + 2*(3.14159 - THETA)*(Y < 0)"
  15  PRINT THETA

       Angle
      0.9273

  16  ENDBREAK
***** break at statement 4 in procedure POLAR
"ENDPROCEDURE"
  17  CALCULATE Deg = THETA*180/3.14159
  18  PRINT Deg

         Deg
       53.13

  19  ENDDEBUG
  20  PRINT Xpos,Ypos,Radius,Angle

        Xpos        Ypos      Radius       Angle
       3.000       4.000       5.000      0.9273
```

5.6 The environment of a Genstat program

The output from the examples in this manual so far was produced in the standard environment. For example, the Genstat statements were not echoed with line numbers when a program was run interactively, but they were when it was run in batch; new lines in the programs were taken as terminators of statements unless a continuation symbol was given; upper-case and lower-case letters were treated as distinct in identifiers. You can change these and other details of the environment of a job by the `SET` directive (5.6.1). It is also possible to find out the current environment, using the `GET` directive (5.6.2). This is of most use inside procedures that are designed to work in a general way.

The definitions of Genstat directives and procedures in this book, and those in the *Genstat Reference Manual*, include details of the default settings of options and parameters. However, you can redefine these defaults at any time with the `SETOPTION` and `SETPARAMETER` directives (5.6.3).

It could be confusing to work with different Genstat environments on different occasions. The ideal way to modify the environment is in a start-up file, which is automatically executed whenever you start using Genstat (5.6.4).

5.6.1 The `SET` directive

`SET` directive

Sets details of the "environment" of a Genstat job.

Options

`INPRINT` = *string tokens*	Printing of input as in `PRINT` option of `INPUT` (`statements`, `macros`, `procedures`, `unchanged`); default `unch`
`OUTPRINT` = *string tokens*	Additions to output as in `PRINT` option of `OUTPUT` (`dots`, `page`, `unchanged`); default `unch`
`DIAGNOSTIC` = *string tokens*	Defines the least serious class of Genstat diagnostic which should still be generated (`messages`, `warnings`, `faults`, `extra`, `unchanged`); default `unch`
`ERRORS` = *scalar*	Number of errors that a job may contain before it is abandoned (0 implies no limit); default is to leave unchanged
`FAULT` = *text*	Sets the Genstat fault indicator (for example, `FAULT=*` clears the last fault); default is to leave the indicator unchanged
`PAUSE` = *scalar*	Number of lines to output before pausing (interactive use only; 0 implies no pausing); default is no change
`PROMPT` = *text*	Characters to be printed for the input prompt; default is to leave unchanged
`NEWLINE` = *string token*	How to treat a new line (`significant`, `ignored`); default is no change
`CASE` = *string token*	Whether lower- and upper-case (small and capital) letters are to be regarded as identical in identifiers (`significant, ignored`); default is no change
`FIELDWIDTH` = *scalar*	Fieldwidth to be used as a default minimum by `PRINT` and other output commands
`SIGNIFICANTFIGURES` = *scalar*	Minimum number of significant figures to be supplied in the default formats determined by `PRINT` and other

	output commands
SEEDS = *pointer* or *scalar*	Defines the current default seeds to be used for random numbers in various parts of Genstat
RUN = *string token*	Whether or not the run is interactive (interactive, batch); by default the current setting is left unchanged
UNITS = *identifier*	To (re)set the current units structure; default is to leave unchanged
BLOCKSTRUCTURE = *identifier*	To (re)set the internal record of the most recent BLOCKSTRUCTURE statement; default is to leave unchanged
TREATMENTSTRUCTURE = *identifier*	To (re)set the internal record of the most recent TREATMENTSTRUCTURE statement; default is to leave unchanged
COVARIATE = *identifier*	To (re)set the internal record of the most recent COVARIATE statement; default is to leave unchanged
ASAVE = *identifier*	To (re)set the current ANOVA save structure; default is to leave unchanged
DSAVE = *identifier*	To (re)set the current save structure for the high-resolution graphics environment; default is to leave unchanged
MSAVE = *identifier*	To (re)set the current save structure for multivariate analysis; default is to leave unchanged
RSAVE = *identifier*	To (re)set the current regression save structure; default is to leave unchanged
TSAVE = *identifier*	To (re)set the current time-series save structure; default is to leave unchanged
VSAVE = *identifier*	To (re)set the current REML save structure; default is to leave unchanged
VCOMPONENTS = *identifier*	To (re)set the current REML model definitions, as specified by VCOMPONENTS and VSTRUCTURE; default is to leave unchanged
WORDLENGTH = *string token*	Length of word (8 or 32 characters) to check in identifiers, directives, options, parameters and procedures (long, short); default * i.e. no change
CAPTIONS = *string tokens*	Controls which captions are displayed (minor, major, meta, unchanged); default unch
TYPESET = *string tokens*	Controls when typesetting commands within textual strings are used (output, graphics); if unset, the existing setting is left unchanged
CMETHOD = *string token*	Controls whether number settings for colour options and parameters are interpreted as RGB values or as numbers of standard colours (rgb, standard); if unset, the existing setting is left unchanged
DATASPACE = *scalar* or *variate*	Updates the current data space allocations; if unset, the existing allocations are left unchanged
WORKINGDIRECTORY = *text*	Sets the working directory; default is to leave this unchanged
ALGORITHMS = *string token*	Controls the use of enhanced computing algorithms (standard, mkl); if unset, the existing setting is left unchanged

ACTIONAFTERFAULT = *string token*	Controls what happens after a fault (continue, stop); if unset, the existing setting is left unchanged
UNSETDUMMY = *string token*	Controls what happens if you specify an unset dummy as the setting of an option or parameter that expects another type of data structure (fault, ignore, warn); if unset, the existing setting is left unchanged
LANGUAGE = *text*	Text with either one or two values to specify a preferred language for output and (optionally) a second choice in case the preferred language is unavailable
YEAR2DIGITBREAK = *scalar*	Controls how 2 digits can be used to specify years
TIMEWITHSECONDS = *string token*	Controls whether seconds are included with the time12 and time24 date representations; (absent, present, unchanged); default unch

No parameters

The default of SET is to do nothing: that is, each option by default leaves the corresponding attribute of the environment unchanged. Of course you have to start somewhere, so an initial environment is defined at the start of any Genstat program; the corresponding initial settings of the options of SET, known as the *initial defaults*, are described below.

The INPRINT option controls what parts of a Genstat job supplied in the current input channel are recorded in the current output file; the input channel can be either an input file or the keyboard. Three parts are distinguished: explicit statements; statements, or parts of statements, that you have supplied in macros using either the ## notation (1.8.2) or the EXECUTE directive (5.4.3); and statements that you have supplied in procedures. The initial default is to record nothing if the output is to the screen, otherwise to record the statements. This aspect of the environment can be modified also by the PRINT option of the INPUT directive (3.4.1) and by the INPRINT option of JOB (5.1.1).

The OUTPRINT option controls how the output from many Genstat directives starts: the output can be preceded by a move to the top of a new page, or by a line of dots beginning with the line number of the statement producing the analysis, or by both. If output is directly to the screen, no new pages are given. The initial default is to give neither if output is to the screen, otherwise to give a new page and a line of dots. Alternatively, this aspect can be modified by the PRINT option of the OUTPUT directive (3.4.3) or by the OUTPRINT option of JOB. The lines of dots are produced by the directives for regression analysis, analysis of designed experiments, REML analysis, multivariate analysis, and time series; also from the FLRV, FSSPM and SVD directives (4.10). If you give an analysis statement within a FOR loop (5.2.1), the line number preceding the line of dots is that of the ENDFOR statement rather than of the analysis statement. New pages are produced with any of the above, and with the GRAPH, HISTOGRAM and CONTOUR directives.

The DIAGNOSTIC option lets you control the level of diagnostic reporting. You might want to do this within a procedure, to prevent faults being reported to a user who does not need to know in detail what is going on inside the procedure. By initial default, all diagnostics - messages, warnings and faults - are printed. You can switch off messages by setting DIAGNOSTIC=warning, or switch off both messages and warnings by setting DIAGNOSTIC=fault. If you set DIAGNOSTIC=*, then no diagnostics will appear. The extra setting gives you extra information, in the form of a dump of the current state of the job; but this is likely to be useful only for developers of Genstat. Printing of diagnostics can also be controlled by the DIAGNOSTIC option of JOB (5.1.1).

The ERRORS option controls what Genstat does when many faults happen within a single job while in batch mode. By initial default, up to five errors per job are reported, and successive faults will not generate diagnostic messages. This ensures, for example, that input intended to be read by a READ statement (3.1.2) will not generate many lines of diagnostics if execution halts because of a fault before the READ statement. Note, however, that this option does not affect the detailed error messages printed by the READ directive itself: these are controlled separately by the corresponding ERRORS option of READ. In interactive mode, the count of errors is restarted after each successful statement is issued, though the option is unlikely to be useful in this mode.

The FAULT option is provided primarily to allow procedure writers to modify the internal record that is kept of the most recent fault indicator. Setting FAULT=* clears the record; you can then use the GET directive (5.6.2) to ascertain whether a fault has occurred since the record was cleared. You can also set the fault indicator to a particular diagnostic, for example

```
SET [FAULT='VA4']
```

A subsequent DISPLAY statement (5.4.1) will then report the chosen fault in the standard way. The fault indicator is automatically cleared at the start of each job.

The PAUSE option lets you specify how many lines of output are produced at a time when you are running Genstat interactively; you might, for example, want to read the output on a terminal screen before more output replaces it. Obviously this is not relevant in batch, and may not be needed in the implementations of Genstat that provide a scrollable output window. The initial default is to send all output to the current output channel as soon as it is available. Some computers can store the output, irrespective of whether Genstat itself has a scrollable window, and let you scroll forward and back to read it at leisure: others just provide keys to freeze the output while you are reading a section, and then to continue to the next segment of output. If you set PAUSE=*n*, then after every *n* lines of output Genstat gives a prompt:

```
*Press RETURN to continue*
```

After you have read the displayed section of output, you can press the <RETURN> key to get the next *n* lines. The counting of lines is restarted each time you give a statement from the keyboard: it is not restarted between separate statements in a macro, procedure or auxiliary input channel. If you have specified that Genstat should echo input lines, these are included among the *n*. Once all the output has been displayed, Genstat prompts for further statements.

The PROMPT option specifies the characters used to prompt for interactive input. The initial default is the greater-than character followed by a space "> ". The prompt can also be modified by the PROMPT option of JOB (5.1.1). Other prompts are used by READ, EDIT, HELP and QUESTION, and these cannot be altered.

The NEWLINE option allows you to cancel the initial default whereby a new line (<RETURN>) is a terminator both for strings within a string list (1.5.2) and for a statement (1.7). Thus, for example, if you specify

```
SET [NEWLINE=ignored]
```

you need no longer use a backslash (\) to continue a statement onto a new line, since <RETURN> is no longer interpreted as the end of a statement. But you will then have to terminate each statement explicitly with a colon.

The CASE option specifies whether upper-case and lower-case letters are to be treated as the same in identifiers (1.4.3). The initial default is that upper and lower case are not the same; thus, an identifier X is distinct from an identifier x. If CASE is set to ignored, then in later statements, both x and X are treated as the same identifier, X. Thus the structure with identifier x cannot be referenced, unless CASE is later reset to significant.

The FIELDWIDTH option allows you to control the minimum fieldwidth that is used as a default by PRINT (3.2) and other output commands. The initial default is 12.

In PRINT the default number of decimal places for a numerical structure is determined by calculating the number that would be required to print its mean absolute value to at least *d* significant figures. The initial default for *d* is four, but you can redefine this using the SIGNIFICANTFIGURES option.

The SEEDS option specifies the default seeds to be used to generate random numbers in various areas of Genstat. You can set SEED to a scalar to define a single seed to be used for all the areas. Alternatively, you can supply a pointer to define a different seed for each area. The elements of the pointer should be labelled to indicate the area concerned: for example 'calculate', and 'randomize' for random-number functions and the RANDOMIZE directive respectively. The easiest way to see the possibilities is to save the current seeds using the SEEDS option of the GET directive; this saves a pointer with elements labelled automatically. You will notice, though, that the GET pointer represents each seed as a variate (with several values) rather than a scalar. This is because, once any randomization has done in an area, there is too much seed information to store in a single number. Variates are equally valid for the elements of the SET pointer. So you can save the current seeds using GET, and then restore them by using the same pointer in SET.

The WORDLENGTH parameter controls the number of characters that are stored and checked in identifiers and names of directives, procedures, options, parameters and functions. In releases prior to 4.2 this was always eight, but from 4.2 onwards you can choose between eight (WORDLENGTH=short) and 32 (WORDLENGTH=long). This can also be controlled by the JOB directive (5.1.1) and, within a procedure, by the PROCEDURE directive (5.3.2). The default is to leave the setting unchanged.

The RUN option controls whether Genstat interprets the program as being in batch or in interactive mode (1.1.1 and 1.1.2); this assumed mode is independent of whether the program really is being run in batch or interactively. Initially, a program is taken to be in interactive mode only if the first input channel and the first output channel are both connected to a terminal. The setting of the assumed mode has two effects - on recovery from faults, and on how HELP (1.2.1) and EDIT (4.7.10) operate.

The UNITS option provides another way of setting the *units structure* in addition to the UNITS directive described in 2.3.4. The setting can be the identifier of a variate or text structure; this will become the default labelling structure of other variates, texts or factors with the same length, in those directives that use such labels. The setting can also be a scalar to specify the default number of units. For further details, see 2.3.4. The setting of the UNITS option is lost at the end of each job within a program.

The BLOCKSTRUCTURE, TREATMENTSTRUCTURE, COVARIATE, ASAVE, DSAVE, MSAVE, RSAVE, TSAVE, VSAVE and VCOMPONENTS options specify special *save structures* (2.9) for graphical and analysis directives. You can set the options only to an identifier that you have previously established by the SPECIAL option of the GET directive (5.6.2) or by the SAVE options in the various analysis directives themselves. For example, if two sets of regression analyses are in progress in one job, the SET directive can be used to switch between them:

```
MODEL [SAVE=S1] Y1
FIT X1
MODEL [SAVE=S2] Y2
FIT X1
SET [RSAVE=S1]
FIT X1,X2
```

This program fits the regression of Y1 on X1, using save structure S1, then the regression of Y2 on X1 with save structure S2. Finally, it fits the regression of Y1 on X1 and X2, because the current regression save structure is changed to S1 before the last FIT statement. The settings of all these options are lost at the end of a job.

The CAPTIONS option controls which captions are displayed by directives and procedures. This can be used inside a procedure to suppress irrelevant captions that would be produced by the procedures or directives that it calls. The setting can be restored by the RESTORE option of the PROCEDURE statement, or by saving the current setting using GET, and then restoring it by using another SET. The initial default is to display all types of caption.

The TYPESET option controls whether typesetting commands within textual strings (see 1.4.2) are recognized used in output and in labels and titles on graphs. The initial default is to use them in both.

The CMETHOD option is useful if you have programs from Release 10 or earlier that use the old way of specifying graphics colours. Prior to Release 11, you had to use one of Genstat's 256 standard colours, and redefine its RGB definition, if necessary, using the COLOUR directive. In Release 11, the representation of colours was changed to allow you to use standard colour names; see 6.9.9 for details. So virtually all options and parameters of the directives and library procedures that define colours were modified to take strings or texts as their settings. Further flexibility was given by interpreting numeric settings directly as RGB values. However, if you have a program from Release 10 or earlier that relies on the old standard colours, you can put

```
    SET [CMETHOD=standard]
```

to interpret numeric settings of colour options and parameters later in your program as standard colour numbers instead of RGB values.

The DATASPACE option allows you to increase the current data space allocations. You can set this to a variate of length three to specify a different size for each of the three types of data: real numbers (for numeric data), integers (for factors and system information) and characters (for texts). Alternatively, you can set it to a scalar to specify the same size for all three types. The sizes are measured in blocks of 32768 values. If any of the data spaces is already larger than the specified size, its size is left unchanged. This option can be useful if you know that your next analysis is likely to require lot of space − it is more efficient to reserve all the space at once, rather than leaving it to Genstat to expand each allocation every time that it becomes full.

The WORKINGDIRECTORY option allows you to set the working directory (the default directory where Genstat will open or save files).

The ALGORITHMS option allows you to request the use of enhanced computing algorithms. The initial default, at the start of any Genstat run is to use only the standard algorithms. However, if you set ALGORITHMS=mkl, it will use algorithms from the Intel® Math Kernel Library for operations such as eigenvalue decompositions and matrix inversion. These should provide much faster performance with large problems.

The ACTIONAFTERFAULT option allows you to control what happens if a fault occurs inside a procedure or during a batch run. The initial default, at the start of any Genstat run, is that execution of the procedure or the batch script stops. However, you can set ACTIONAFTERFAULT to continue to request that it continues instead. The FAULT option of GET (5.6.2) can be used to access the most recent fault code, so that you can make your own decision about what to do next if a fault occurs.

A dummy (2.2.2) is a data structure that stores the identifier of a data structure. This can be useful with options and parameters that expect another type of data structure. If you supply a dummy, it will be replaced by the identifier that it stores. The UNSETDUMMY option controls what happens if the dummy is unset. The initial default is to give a warning, and replace the dummy by the default for the option or parameter if one has been defined, or otherwise to treat the option or parameter as though it had not been set. If you set UNSETDUMMY to ignore, no warning is given. Finally, if you set it to fault, unset dummies are treated as faults.

Some Genstat commands can now provide output in languages other than English. The LANGUAGE option allows you to supply a text with either one or two values to specify your preferred language in its first value, and (optionally) your second choice in its second value. Output will then be generated in your preferred language if that is available. Otherwise it may be in your second-choice language or, if neither are available, the command will generate the ordinary English output.

The YAR2DIGITBREAK option controls how two digits can be used to specify years: whether these represent years in the 1900's or the 2000's. YAR2DIGITBREAK specifies the cut-off date: dates less than this value represent years beginning 20, and two digit dates greater than or equal to this value will represent years beginning 19. For example, if it is set to 30, years in the range 00 - 29 will represent the years 2000 - 2029 and years in the range 30 - 99 represent the years

1930 - 1999. Alternatively, the value 0 ensures that all 2 digit dates belong to the 1900's, while the value 100 means that they all belong to the 2000's.

The `TIMEWITHSECONDS` option controls whether seconds will be present or absent in output with the `time12` and `time24` date representations. The default is to leave the current setting unchanged.

5.6.2 The `GET` directive

The `GET` directive allows you to access the current settings of the environment. This can be particularly useful in procedures, when details of the environment may need to change and be reset later to their original state. Sometimes it may be sufficient just to use the `PRESERVE` option of the `PROCEDURE` directive (5.3.2) for this purpose, but this causes them to be reset only at the end of a procedure.

`GET` directive

Accesses details of the "environment" of a Genstat job.

Options

`ENVIRONMENT` = *pointer*	Pointer given unit labels `'inprint'`, `'outprint'`, `'diagnostic'`, `'errors'`, `'pause'`, `'prompt'`, `'newline'`, `'case'`, `'run'`, `'wordlength'`, `'captions'`, `'typeset'`, `'cmethod'`, `'dataspace'`, `'algorithms'`, `'actionafterfault'`, `'unsetdummy'`, `'language'`, `'year2digitbreak'` and `'timewithseconds'` to save the current settings of those options of `SET`; default `*`
`SPECIAL` = *pointer*	Pointer given unit labels `'units'`, `'blockstructure'`, `'treatmentstructure'`, `'covariate'`, `'asave'`, `'dsave'`, `'msave'`, `'rsave'`, `'tsave'`, `'vsave'` and `'vcomponents'`, used to save the current settings of those options of `SET`; default `*`
`LAST` = *text*	To save the last input statement; default `*`
`FAULT` = *text*	To save the last fault code; default `*`
`FIELDWIDTH` = *scalar*	Saves the fieldwidth currently defined as the default minimum for `PRINT` and other output commands
`SIGNIFICANTFIGURES` = *scalar*	Saves the minimum number of significant figures currently to be supplied in the default formats determined by `PRINT` and other output commands
`SEEDS` = *pointer*	Saves a pointer to variates defining the seeds currently used as defaults by random-number functions, the `RANDOMIZE` directive, and internally by various other directives
`EPS` = *scalar*	To obtain the value of the smallest x (on this computer) such that $1+x > 1$; default `*`
`NJOB` = *scalar*	Number of the current job within the program; default `*`
`VERSION` = *pointer*	Information about the version of Genstat that is being used; default `*`
`PID` = *scalar*	Gets an integer value unique in the current job to use, for example, in names of temporary files
`WORKINGDIRECTORY` = *text*	Saves the name of the current working directory

No parameters

The ENVIRONMENT and SPECIAL options of GET are used to access and save the current settings of the options of the SET directive (5.6.1). The options of SET are divided into two groups. Those that apply to the general environment can be saved using the ENVIRONMENT option: these are INPRINT, OUTPRINT, DIAGNOSTIC, ERRORS, PAUSE, PROMPT, NEWLINE, CASE, RUN, WORDLENGTH, CAPTIONS, TYPESET, CMETHOD, DATASPACE, ALGORITHMS, ACTIONAFTERFAULT, UNSETDUMMY, LANGUAGE, YEAR2DIGITBREAK and TIMEWITHSECONDS. Those that apply only to the save structures associated with particular directives are saved by using the SPECIAL option: these are UNITS, BLOCKSTRUCTURE, TREATMENTSTRUCTURE, COVARIATE, ASAVE, DSAVE, MSAVE, RSAVE, TSAVE, VSAVE, VCOMPONENTS.

When you use the ENVIRONMENT option, Genstat sets up a pointer (2.6) with units identified by the labels of the corresponding options of SET: these labels are 'inprint', 'outprint', and so on. These labels can be specified in either lower or upper case, or any mixture. Each unit of this pointer contains one or more strings, or a scalar, to represent the current setting. Thus, the statement

```
GET [ENVIRONMENT=Env]
```

would set up a pointer called Env with elements Env['inprint'], Env['outprint'], and so on. Each element can also be referred to by its position in the pointer; for example, Env['inprint'] is the same as Env[1]. Example 5.6.2 shows what Env would contain in a batch run where the options of SET had not been changed from their default values.

Example 5.6.2

```
   2  GET [ENVIRONMENT=Env]
   3  PRINT [RLWIDTH=24; ORIENT=across; SQUASH=yes] Env[]; FIELD=16
          Env['inprint']      statements
         Env['outprint']            dots            page
       Env['diagnostic']        messages        warnings          faults
           Env['errors']               5
            Env['pause']               0
           Env['prompt']               >
          Env['newline']     significant
             Env['case']     significant
              Env['run']           batch
       Env['wordlength']            long
         Env['captions']           minor           major            meta
          Env['typeset']          output        graphics
          Env['cmethod']             rgb
        Env['dataspace']               1               1               1
       Env['algorithms']        standard
 Env['actionafterfault']            stop
       Env['unsetdummy']            warn
         Env['language']
  Env['year2digitbreak']              30
```

Thus you do not have to know how the environment has been set in order to change it and then restore it; you can use GET to find out about it, and SET to change it back. For example, suppose that you wanted to stop temporarily the echoing of statements to the output file in a batch program. In the following program the first SET statement cancels the echoing, if indeed any echoing is in progress, and the second restores echoing to what it was before the first SET.

```
GET [ENVIRONMENT=Env]
SET [INPRINT=*]
```

(more statements)

```
SET [INPRINT=#Env['inprint']]
```

The SPECIAL option similarly sets up a pointer to save its information. The labels of the pointer are 'units', 'blockstructure', and so on. These can again be specified in either lower or upper case, or any mixture. The first element of the pointer is the units structure, or, failing that, the number of units if you have defined it for the current job. Printing the contents of the other elements is not usually informative, as the information is stored in coded form. The last ten elements of the pointer allow you to access the special save structures defined by graphical and analysis directives. They are most useful for recovering information about an analysis when you have not already specified an explicit save structure. (Otherwise you would have to do the analysis all over again.) For example, in the statements at the end of 5.6.1, if you had not set the SAVE option in the first MODEL statement, you could instead put

```
MODEL Y1
FIT X1
GET [SPECIAL=S1]
MODEL [SAVE=S2] Y2
FIT X1
SET [RSAVE=S1['rsave']]
FIT X1, X2
```

The SPECIAL option of GET also allows you to access the save structures associated with the analysis-of-variance directives BLOCKSTRUCTURE (2:4.2.1), COVARIATE (2:4.3.1) and TREATMENTSTRUCTURE (2:4.1.1). This facility is used by the ASTATUS procedure (2:4.9.1), which may provide a more convenient way of accessing these structures.

The LAST option is used to save the latest statement that you have input. You can then give the statement again later in the job without having to retype it, though some implementations of Genstat provide a simpler recall facility using the cursor keys. The option has the same effect as setting up a macro (1.8.2) containing a single statement, and is accessed in the same way. For example, the statements

```
PRINT [SERIAL=yes; IPRINT=*; SQUASH=yes] !t('New Data'),Y
GET [LAST=Prdat]
```

(statements)

```
READ Y
```

(data)

```
##Prdat
```

would print the data, Y, under the title New Data and save the PRINT statement in a text called Prdat. After the next data set is read, the heading New Data and the new data set are printed in the same format as the previous data set. (The options of PRINT are described in 3.2.1.)

The FAULT option is used to save the last fault code as a single string in a text structure. (A complete list of fault code definitions is available from the HELP environment facility.) This option is particularly useful in procedures, in combination with the DIAGNOSTIC and FAULT options of SET, to control the printing of diagnostics.

The FIELDWIDTH option saves the fieldwidth currently defined as the default minimum for PRINT and other output commands, and the SIGNIFICANTFIGURES option saves the minimum number of significant figures currently to be supplied in the default formats determined by PRINT and other output commands (3.2).

The SEEDS option saves a pointer containing variates, each containing four values, which define the seeds currently used as defaults by random-number functions, the RANDOMIZE directive, and internally by various other directives. The pointer elements are labelled to identify the use of the seeds concerned: for example 'calculate', and 'randomize' for random-number functions and the RANDOMIZE directive respectively.

The EPS option is used to obtain the smallest number, ε, such that 1.0+ε is recognized by your computer to be greater than 1.0; this is an indication of the precision of the computer, which can affect the behaviour of some of the algorithms used by Genstat. `EPS` can be used, for example, when testing for convergence of iterative algorithms.

The `NJOB` option provides the current job number within the Genstat program. It is used in the start-up file (5.6.4) to distinguish between statements to be executed just at the start of the program, and those to be executed at the start of each job.

The `VERSION` option provides information about the version of Genstat that is being used. This is particularly useful within general programs or procedures. It saves a pointer containing the following elements.

`release`	is a scalar storing the release number, for example 21.10. The main information is in the integer part and the first decimal place; the second decimal may be used to distinguish between sub-releases with minor changes or corrections.
`patch`	shows whether the release includes a patch.
`build`	is the build number (useful for support).
`implementation`	identifies the type of computer for which the version has been implemented, for example `'PC'`.
`system`	indicates the operating system, for example Windows 11.
`version`	may contain further information relevant to particular implementations.
`description`	gives the name of the release, for example Genstat Twenty-first Edition.
`bits`	gives the number of bits for which the implementation has been built, for example 64 for a 64-bit version.

The `PID` option saves a scalar containing an integer value that is unique within the current job. You might want to use this, for example, to define a unique name for a temporary file.

The `WORKINGDIRECTORY` option saves a text containing the name of the current working directory.

5.6.3 Changing the defaults of options and parameters

`SETOPTION` directive

Sets or modifies defaults of options of Genstat directives or procedures.

Option

`DIRECTIVE` = *string token*	Directive (or procedure) to be modified

Parameters

`NAME` = *string tokens*	Option names
`DEFAULT` = *identifiers*	New default values

`SETPARAMETER` directive

Sets or modifies defaults of parameters of Genstat directives or procedures.

Option

`DIRECTIVE` = *string token*	Directive (or procedure) to be modified

Parameters

`NAME` = *string tokens*	Parameter names
`DEFAULT` = *identifiers*	New default values

These directives change the defaults settings for the specified directive or procedure for the remainder of the current job. If you use one of these directives in your start-up file (5.6.4) you can make the changed default apply in all your use of Genstat.

To achieve any effect, the option and both parameters of either of these directives must be set. The `DIRECTIVE` option specifies the name of the directive or procedure that is affected, and the `NAME` parameter indicates the option or parameter whose default is to be changed. The settings are strings, so need not be quoted because all directive and procedure names are valid as unquoted strings. The `DEFAULT` parameter is then set to a data structure to provide the new default that you want to be assumed. For example, the following statement modifies the `PRINT` option of the `FIT` directive which carries out regression analysis.

```
    SETOPTION [DIRECTIVE=FIT] PRINT; DEFAULT='deviance'
```

The usual default of the `PRINT` option in `FIT` is to print a statement of the model, a summary of the analysis, and the parameter estimates: this corresponds to the setting `PRINT=model,summary,estimates`. This `SETOPTION` statement therefore redefines the default so that any subsequent `FIT` statement in the job will report only the residual deviance unless you explicitly set the `PRINT` option.

The defined mode of the `PRINT` option of `FIT` is "strings". However, the `DEFAULT` parameter of `SETOPTION` expects a data structure (to allow for all the other modes that might occur), and so it must be set to a text structure containing the string (or strings) that you want to be the default. Similarly, if the defined mode of the option or parameter is "numbers", "expression" or "formula", you must supply a variate, an expression structure or a formula structure containing the new default. If the defined mode is "identifier", the setting of `DEFAULT` is simply an identifier, which must be of the required type if this is specified in the definition of the directive or procedure.

To reset the `PRINT` option of `FIT` back to its usual default, you would need to give the statement

```
    SETOPTION [DIRECTIVE=FIT] PRINT;\
      DEFAULT=!t(model,summary,estimates)
```

The `SETOPTION` and `SETPARAMETER` directives can also be used to change defaults of any procedure: this may be a procedure in the standard Procedure Library, the Site Library, or a personal library that you have already opened in the current program, or it may be a procedure that you have defined explicitly in the job.

For example, we shall modify the default action of the `DESCRIBE` procedure in the standard Library. This procedure prints a summary of values in a variate, and by default does not store the summary. It has a parameter called `SUMMARIES` which you can set if you want the summaries to be stored. The statement

```
    SETPARAMETER [DIRECTIVE=DESCRIBE] SUMMARIES; DEFAULT=Sum
```

would change the default action of this procedure, so that after using it without setting the `SUMMARIES` parameter the summaries would be available in variate `Sum`.

5.6.4 Start-up files

A start-up file contains Genstat statements that are to be executed at the beginning of every job. Thus in an interactive run they are executed before Genstat prompts you for commands, and in batch before Genstat executes the statements that you have prepared. The standard start-up file, distributed with Genstat, performs two tasks: it prints a banner describing the version of Genstat that is being used, and it opens a file to keep a record of interactive sessions. You can set up your own start-up file and arrange for it to be executed instead, to define your preferred Genstat environment automatically at the start of each job.

The standard start-up file contains job-control structures to allow separate sets of statements to be executed in batch and interactive modes, and at the start of the first job and subsequent jobs in the program. In fact, little is done in batch mode because neither of the tasks listed above is relevant: the banner is designed to help interactive users, and there is no need to keep a record of statements. When running interactively, the record file is opened at the start of the first job; once open, there is no need to open it at the start of subsequent jobs. Similarly, the banner is printed just at the start of the first job.

You can take a copy of the standard file and edit it – perhaps just to remove the banner, or perhaps to insert a `SET` statement (5.6.1) to change the environment. If you prefer an alternative default for an option or parameter of a directive that you use frequently, you might want to insert a `SETOPTION` or `SETPARAMETER` statement (5.6.3); if so, you must put it into the first section of the file, which is executed in both modes and at the start of all jobs. You might also include an `OPEN` statement (3.3.1) to provide automatic access to a personal procedure library or backing-store file; or you could define macros to carry out operations you require frequently.

Having created your own start-up file, you can arrange for it to be used in place of the standard one by following the instructions given in the local documentation.

5.7 Communicating with other programs

Genstat is designed as a general statistical package, and so contains facilities for most of the statistical methods that you may need; but there are many other possible requirements. Some of these will use information that you can produce with Genstat; others will generate information that you can analyse with Genstat. Therefore you may need to connect Genstat to other programs.

The simplest method of communication between programs is via files. One program might extract data and store it in a file – either in character form (3.2) or in binary (3.7); this program might be written in the data-base language. A second program, written in the Genstat language, might then read that file, process the data, and perhaps form another file; and so on.

If you are content to work step-by-step, first running one program then another, you can simply use the facilities described in Chapter 3 for storing and accessing information in files. However, you may prefer to run programs concurrently. The `SUSPEND` directive, described in 5.7.1, allows you temporarily to halt Genstat and to perform other tasks on the computer before continuing to run Genstat. You can arrange to communicate information between Genstat and other programs that are run while Genstat is halted, simply by reading and writing files.

The `PASS` directive, described in 5.7.2, works like `SUSPEND`, but is designed to deal automatically with the transfer of information between Genstat and separate programs. You can set up the programs using Fortran, incorporating a Fortran subprogram that is distributed with Genstat to deal with communication, or use some other computing language able to read the data file used to transfer the information.

5.7.1 Suspending Genstat to give commands to the operating system

`SUSPEND` directive

Suspends execution of Genstat to carry out commands in the operating system; this directive may not be available on some computers.

Options

`SYSTEM` = *text*	Commands for the operating system; default: prompt for commands (interactive mode only)
`CONTINUE` = *string token*	Whether to continue execution of Genstat without waiting for commands to complete (`yes`, `no`); default `no`
`MINIMIZE` = *string token*	Whether to minimize the console window (`yes`, `no`);

	default `no`

No parameters

If you run the command

```
    SUSPEND
```

(with no options) in Genstat *for Windows*, a command window will open, in which you can enter commands in the usual way. You can return to Genstat by closing the window (e.g. by typing `EXIT`).

You can use the `SYSTEM` option to specify the commands to run in the operating system. By default, Genstat then pauses while the commands run, and continues after they finish. This provides a convenient way to run an external program. For example, it is used by the `_CDCALL` procedure, included with `CDNBLOCKDESIGN`, to run the CycDesigN engine. `CDNBLOCKDESIGN` (and other `CDN` procedures that use `_CDCALL`) use the `OPEN` directive to open a file to contain the data for the engine, and the `PRINT` directive (with the `CHANNEL` option set to the filename) to form its contents. `_CDCALL` uses `TXCONSTRUCT` to construct the command for `SUSPEND`. The CycDesigN engine writes its output to another file, which can be read afterwards by `CDNBLOCKDESIGN` (or the other `CDN` procedures).

You can set option `CONTINUE=yes` if the operating-system commands do not need to run before your subsequent Genstat commands. For example, you might simply want to post a message to say that your Genstat run is executing.

You can set option `MINIMIZE=yes` to minimize the console window in which the comands run.

5.7.2 Executing external programs

On some computers, you can arrange that one program, such as Genstat, calls for another to be executed, with information passed directly between the two. You can thus cause Genstat to execute your own subprograms without having to modify Genstat in any way. You can do this with the `PASS` directive. To find out if the `PASS` directive has been implemented in your version, you can either look at local documentation, or issue the `PASS` command with no options or parameters. If it is not available, you will get a message saying that `PASS` has not been implemented. You could then use the `SUSPEND` directive instead.

To use the `PASS` directive, you must first compile a program library file (DLL on Windows or SO on Linux) of your own code. You will need to program the reading and writing of the data to communicate with Genstat within the functions that are called. The details are explained below.

`PASS` directive

Performs tasks specified in subprograms supplied by the user, but not linked into Genstat; this directive may not be available on some computers.

Option

`NAME` = *text*	Filename of external executable program; default `'GNPASS'`

Parameters

`DATA` = *pointers*	Structures whose values are to be passed to the external program, and returned
`ERROR` = *scalars*	Reports any errors in the external program

The NAME option specifies the file name of a program library (DLL for Windows or SO for Linux) and a function name within the library, separated by a dollar symbol $. If the file extension is missing, .DLL will be appended for Windows and .SO will be appended for Linux. An advantage of omitting a file extension is that,if you have libraries that just differ in their extensions, the same command could be run on Windows and Linux. If no path is provided for the library, the system will search the Genstat Bin folder first, and then the folders in the PATH environment variable. An FI 20 fault will be given if the library is not found. The library is then searched for an exported function with the name specified after the dollar (this is case sensitive). If no function name is provided, the function GNPASS is used. An FI 21 fault is given if the function is not found.

You can use the DATA pointer to pass the values of any data structures except texts. All the structures needed by your subprograms must be combined in a pointer structure, unless only one structure is needed. The structures must have values before you include them in a PASS statement; if you want to use some of the structures to store results from your subprograms, you must initialize them to some arbitrary values, such as zero or missing. If you specify several pointers in a PASS statement, your subprograms will be invoked several times, to deal in turn with the set of structures stored by each pointer. However, the values of the structures in all the pointers are copied before any work is done by your subprograms. Thus, if you want to operate with PASS on the results of a previous operation by PASS, you must give two PASS statements with one pointer each rather than one statement with two pointers. The ERROR parameter can pass a scalar value back into Genstat to indicate whether any errors have occurred.

As an example, consider using PASS to carry out a simple transformation of a variate, as would be done by the statement

```
    CALCULATE W = M*(V+S)**2
```

where V and W are variates, and M and S are scalars. Example GNPASS programs which calculate this transformation in Fortran (GENPASS.f90) and C (GNPASSC.c) are available is the Source directory of the Genstat installation. The functions in Fortran must have arguments (INFILE, OUTFILE) of type CHARACTER(*) and in C (INFILE, OUTFILE, INLEN, OUTLEN) with the first two arguments of type char * and the last two of type int. INFILE gives the filename of the input data and OUTFILE gives the filename of the output to be read into Genstat. The extra C arguments are the lengths of the first two arguments. The example files contain a function SQUARE that calculates the transformation. The code needs to handle missing values by comparing data with the missing value indicators given in INFILE. Compiled versions of these for Windows (GNPASS.dll and GNPASSC.dll) are also in the Source folder. To create your own library you can use these as a template, replace SQUARE with your own function, and then compile and link the code into a program library. To use the Fortran example, run the following statements:

```
    SCALAR S,M; VALUE=2,10
    VARIATE V,W; VALUES=!(1...10),!(10(*))
    TEXT Lib; VALUE='%GENDIR%/Source/GNPASS$GNPASS'
    PASS [NAME=Lib] !p(V,S,M,W)
```

The PASS statement causes the program to run, and assigns the calculated values to the variate W. To use the C program you would use GNPASSC as the library.

Numbers can be used in place of scalars, as usual in Genstat statements:

```
    PASS [NAME='%GENDIR%/Source/GNPASS$GNPASS'] !P(V,2,10,W)
```

To transform the values in both V, as above, and another variate X, with values 10...50 say, you could give the extra statements:

```
    VARIATE X,Y; VALUES=!(10...50),!(50(*))
    PASS [NAME=Lib] !p(V,2,10,W),!p(X,2,10,Y)
```

After preparing the Fortran or C program, you need to form it into an executable program, using a Fortran or C compiler. It may also be possible to use other source languages, provided the input and output formats of their compilers are compatible with that used by Genstat. All floating point values are passed from Genstat as type `REAL*8` for Fortran or `double` for C. Factors and other items are passed as `INTEGER` for Fortran or `int` (4 bytes) for C. The `GNPASS` programs loop around the pointers, with the number of pointers (an integer) provided as the first value in the `INFILE` file. The next 3 items in the input file are the missing value representation for reals (given twice for historical reasons) and integers. Then there are sets of values for each pointer: the number of structures passed, and then the lengths of the arrays of type double precision, single precision (this is not used and should be zero) and integer. Then, for each structure in turn, the file contains its length, mode, Genstat origin and maximum block size (all integers) and its data. The Genstat origin is not used within the program, but should be written back to the result file. The maximum block size is no longer needed, as the files are not now written with a record structure but in unformatted binary mode. However, it must also be written to the result file. The Fortran program reads and writes the data in blocks, but the C program just uses single statements for this. The data are in double precision for mode 2 or integer for mode 3. The results must be written to the `OUTFILE` file in the same format, other than that the number of pointers is replaced by an integer error code (0 for success) which is returned in the `ERROR` parameter, and the missing values indicators are omitted. The example programs read the real and integer data into a single array with a calculated offset for each structure, and pass this in a common block/global structure. However, the data could be saved into individual structures and passed as arguments to the subroutine in your own program.

5.7.3 Executing external functions

You can also call external programs through the `OWN` function. The function call has the form

```
OWN(x; 'func'; p1; p2...pn)
```

where `x` is the input argument. The name of the function (here `func`), the number of additional parameters (*n*) and the DLL that contains the function must be defined, in advance, by the `EXTERNAL` directive as explained below.

EXTERNAL directive

Declares an external function in a DLL for use by the OWN function.

Options

`LIBRARY` = *text*	Name of DLL file containing the function

Parameters

`FUNCTION` = *text*	Name of the function entry point in the DLL
`NAME` = *text*	Name for the function to be used in the `OWN` function; default uses the name set in `FUNCTION`
`RESULTS` =*string token*	The type of result returned from the function (`summary`, `transformation`); default `tran`
`NPARAMETERS` = *scalar*	The number of parameters in the function call; default 0
`ERRORS` = *scalar* or *variate*	Error codes returned from the function; default `*` i.e. no error codes
`MESSAGES` = text	Messages for the corresponding error codes

To use external functions, you first need to create their DLLs by compiling Fortran or C programs. You can then use `EXTERNAL` to define their links to Genstat. For example:

```
EXTERNAL [LIBRARY='CurveFuncs.dll'] FUNCTION='PCNORMAL'; NPAR=0
CALCULATE EX = OWN(X; 'PCNORMAL')
EXTERNAL [LIBRARY='CurveFuncs.dll'] FUNCTION='PEAKRISE'; NPAR=1
CALCULATE P = OWN(X; 'PEAKRISE'; 10)
EXTERNAL [LIBRARY='CurveFuncs.dll'] FUNCTION='HWA';\
   RESULTS=summary; NPAR=5
CALCULATE SS = OWN(X; 'HWA'; a; b; g; n; s)
```

defines three functions, all in the `CurveFuncs.dll`, and then uses these in calculations. The first two functions return variates with the same length as the first argument `X`, and the third returns a scalar. They have zero, one and five parameters respectively. The parameters in the `OWN` function follow the data and name arguments, and must be scalars.

The `LIBRARY` option specifies the name of the file. If the full path to the DLL is not provided, the user add-ins folder is searched first. If the file is not found there, the system add-ins folder is searched. An `FI 11` fault is generated if the file is not found.

The `FUNCTION` parameter gives the name of the entry point in the DLL for the function, and is case insensitive. The entry point must be exported when the program library is compiled. For example, in a C program the function declaration should contain `__declspec(dllexport)` or its equivalent. If the function entry point is not found in the program library, an `FI 12` fault is generated. If you wish to refer to the function in the `OWN` function by a different name to its entry-point name, you can define that name with the `NAME` parameter.

The `RESULTS` parameter indicates what type of result is returned: `summary` returns a scalar and `transformation` (default) returns a structure of the same type and size as the first argument.

The `NPARAMETERS` parameter defines the number of scalar parameters that follow the function name in the `OWN` function. By default there are none.

The `ERRORS`, and `MESSAGES` parameters can be used to set up user-defined fault codes and text for the corresponding error messages for the external functions. If you are using several external functions in a program, you should use different error codes in the different functions, unless the meaning of the code is common to all functions, as the error code/message table is combined over all functions. This allows you to specify just one set of error codes/messages over a set of functions in a library. So, for example, all functions could return a common code 9 if they run out of memory.

The function declaration in C takes the form:

```
long NAME(double* X, int* NX, double* P, int* NP, double* R, int* NR)
```

where

`X`	is an array of the input data from the first argument in the `OWN` function,
`NX`	is the number of elements in `X`,
`P`	is a pointer to an array of the parameters in `OWN` function,
`NP`	is the number of parameters,
`R`	is a pointer to the array to hold the results, and
`NR`	is the number of elements in the result array.

`NR` must be 1 if `RESULTS = summary` and `NX` otherwise. The passed array `P` is `NP+1` long, and the last element is the value that Genstat uses to represent a missing value. The function should return zero if it completes successfully, and a positive error code if there is a fault. The error code will be translated to the text in `MESSAGES` if the error code matches one set up by `ERRORS`.

For a Fortran program, the declaration would be:

```
INTEGER FUNCTION NAME(X,NX,P,NP,R,NR)
   INTEGER   NX,NP,NR
   REAL*8    X(*),P(*),R(*)
   !DIR$ ATTRIBUTES REFERENCE X,NX,P,NP,R,NR
```

There are example programs in Fortran (`OwnFunction.f90`) and C (`OwnFunction.c`) in the Genstat Source folder. These can be compiled to a DLL library using the appropriate compiler and linker settings. A DLL created from these (`OwnFunction.dll`) is included in the Genstat system AddIns folder should you want to test this. The function `OwnFunction` in this DLL calculates the sine of an argument in degrees rather than the usual radians in the standard `SIN` function. Note, however that these simple examples do not test for missing values in the input data.

Any numeric structure (scalar, variate, matrix, symmetric matrix or diagonal matrix) can be passed to the `OWN` function and, when `RESULTS = transformation`, the returned result will be the same type and size of structure (e.g. passing a 3 by 4 matrix will return a 3 by 4 matrix). If you need to pass multiple variates to the function, these could be stacked by the `APPEND` procedure, and the number of variates could be passed as a parameter. Within the function you would extract the stacked values back into 3 arrays and process these individually. If a single variate is to be returned, you would set just the first `NX` values in `R`, and then use its first `NX` values in Genstat to create the resulting variate. For example:

```
EXTERNAL [LIBRARY=VarFunc.dll] FUNCTION='XYZFUNC'; NPAR=1
APPEND [V3] X,Y,Z
CALCULATE V = OWN(V3; 'XYZFUNC'; 3)
CALCULATE N = NVALUES(X)
CALCULATE R = V$[!(1...N)]
```

6 Graphical display

Genstat can produce graphical output in two distinctively different styles. These are *line-printer* graphics and *high-resolution* graphics. As the name suggests, line-printer graphics are designed for printing on ordinary printers, and are also suitable for display on terminals and PC screens. Thus no special equipment is required; also the plots form an integral part of the Genstat output, and can thus be interspersed with other results during the analysis of the data. High-resolution graphics provide a more attractive alternative. Lines and points are plotted with far greater precision, and a wider range of plotting symbols can be used to enhance the output. Also most devices allow the use of colour. A wide range of plots can be produced: graphs and histograms in two or three dimensions, contour plots, shade diagrams, three-dimensional surfaces and pie charts. High-resolution graphics can be produced interactively on graphics terminals, workstations or PC screens. Some of these support graphical input, which can be used for example to allow interactive identification of outliers. Plots can be also saved in files using standard formats that are suitable for plotters or laser printers or for importing into word-processed documents.

Genstat *for Windows* has a selection tool that allows you to select a high-resolution graph and customize its appearance. You can also modify many aspects of the graph, such as colours, line styles, plotting symbols, fonts and axes, interactively after it has been plotted. However, even here you may find it useful to know the commands, in case you want to study the input log or to develop new types of display.

The directives for high-resolution graphics have two main purposes. There are those that define the "graphics environment" for subsequent plots, and those that do the plotting. The default environment, set up at the start of a program, will often be satisfactory. However, you can modify the environment to customize the plots using the following commands:

`DEVICE`	switches between graphics devices (6.9.1)
`FRAME`	defines the positions and appearance of the plotting windows within the graphics frame (6.9.3)
`FFRAME`	forms multiple windows in a plot-matrix for high-resolution graphics
`XAXIS`	defines the x-axis in a graphical window (6.9.4)
`YAXIS`	defines the y-axis in a graphical window (6.9.5)
`ZAXIS`	defines the z-axis in a graphical window (6.9.6)
`AXIS`	defines an oblique axis for high-resolution graphics (6.9.7)
`PEN`	defines properties of graphics "pens" (6.9.8)
`GETRGB`	provides a standard sequence of colours, defined by the initial defaults of the Genstat pens (6.9.9)
`DCOLOURS`	forms a band of graduated colours for graphics (6.9.9)
`DFONT`	defines the default graphics font (6.9.12)
`DHELP`	provides information about the graphics environment (6.9)
`DKEEP`	copies details of the graphics environment into Genstat data structures (6.9.10)
`DLOAD`	loads the graphics environment settings from an external file (6.9.11)
`DSAVE`	saves the current graphics environment settings to an external file (6.9.11)

The directives for plotting high-resolution graphs are:

`DGRAPH`	produces scatter plots and line graphs (6.2.1)
`D3GRAPH`	plots a 3-dimensional graph (6.2.2)

`DHISTOGRAM`	plots histograms (6.3.1)
`BARCHART`	plots bar charts (6.3.2)
`DCONTOUR`	plots contour maps (6.4.1)
`DSHADE`	plots a shade diagram of three-dimensional data (6.4.2)
`DSURFACE`	draws a perspective plot of a two-way array of numbers (6.4.3)
`D3HISTOGRAM`	plots three-dimensional histograms (6.4.4)
`DBITMAP`	plots a bit map of RGB colours (6.5)
`DPIE`	plots pie charts (6.6.1)
`DCLEAR`	clears a graphics screen (6.8.1)
`DSTART`	starts a sequence of related plots (6.8.2)
`DFINISH`	ends a sequence of related plots (6.8.2)
`DDISPLAY`	redraws the current graphical display (6.9.2)

You can add arrows, annotation and reference lines to graphs:

`DARROW`	adds arrows to an existing plot (6.7.3)
`DTEXT`	adds text to a graph (6.7.1)
`DFRTEXT`	adds text to the graphics frame
`DREFERENCELINE`	adds reference lines to a graph (6.7.2)

Some implementations support interactive graphics devices that allow information to be read from the screen:

`DREAD`	reads locations of points from an interactive graphics device

Other facilities are provided by procedures in the `graphics` module of the Library:

`BANK`	calculates the optimum aspect ratio for a graph
`BOXPLOT`	draws box-and-whisker diagrams (2:2.2.2)
`DARROW`	adds arrows to an existing plot (1:6.7.3)
`DERRORBAR`	adds error bars to a graph (1:6.7.4)
`DELLIPSE`	draws a 2-dimensional scatter plot with confidence, prediction and/or equal-frequency ellipses superimposed
`DKEY`	adds a key to a graph (1:6.7.5)
`DTEXT`	adds text to a graph (1:6.7.1)
`DFRTEXT`	adds text to the graphics frame
`DREFERENCELINE`	adds reference lines to a graph (1:6.7.2)
`DCOMPOSITIONAL`	plots 3-part compositional data within a barycentric triangle
`DMASS`	plots discrete data like mass spectra, discrete probability functions
`DOTPLOT`	displays a dot-plot (2:2.2.6)
`DPARALLEL`	displays multivariate data using parallel coordinates (2:2.7.2)
`DPROBABILITY`	plots probability distributions, and estimates their parameters (2:2.2.7)
`DRESIDUALS`	produces model-checking plots of residuals
`DMSCATTER`	displays a scatter-plot matrix (6.8.4)
`DSPIDERWEB`	displays spider-web and star plots
`DTIMEPLOT`	produces horizontal bars displaying a continuous time record
`DXDENSITY`	produces one-dimensional density (or violin) plots
`DXYDENSITY`	produces density plots for large data sets (6.4.5)

DXYGRAPH	draws two-dimensional graphs with marginal distribution plots alongside the y- and x-axes
DYPOLAR	produces polar plots
RUGPLOT	draws "rugplots" to display the distribution of one or more samples (2:2.2.3)
STEM	plots a stem-and-leaf chart (2:2.2.4)
TRELLIS	produces trellis plots for each level of one or more factors (6.8.3)

With line-printer graphics, the standard character set, made up of letters, digits and punctuation characters, is used to produce a graphical representation of the data. This will be of *low resolution*, typically 24 rows by 80 columns for screen display, 132 by 48 or 80 by 60 for a printer; but this is often adequate for a quick assessment of the data, or for checking the assumptions of an analysis. Histograms, graphs and contour plots can be produced in this basic style. The relevant directives are:

LPGRAPH	produces scatter plots and line graphs (6.10.1)
LPHISTOGRAM	plots histograms (6.10.2)
LPCONTOUR	plots contour maps of two-way arrays of numbers (6.10.3)

6.1 Introduction to high-resolution graphics

The `DGRAPH` directive is used in this section to introduce the structure of the high-resolution graphics in Genstat. A full description of `DGRAPH` is given in Section 6.2.1.

Before producing any high-resolution plots, you must first select an appropriate output *device*. This can either be screen-based, for interactive use, or it may send the output to a file in one of a number of standard formats suitable for plotters, printers or word-processed documents. Different versions of Genstat may support different types of device; inevitably there are minor differences in the details of their operation, and these are discussed further in the description of the `DEVICE` directive (6.9.1). The default graphics device is chosen to be the most appropriate for each version and, if this is suitable, no explicit action is required before you start plotting. For Genstat *for Windows*, this is the Genstat Graphics Viewer.

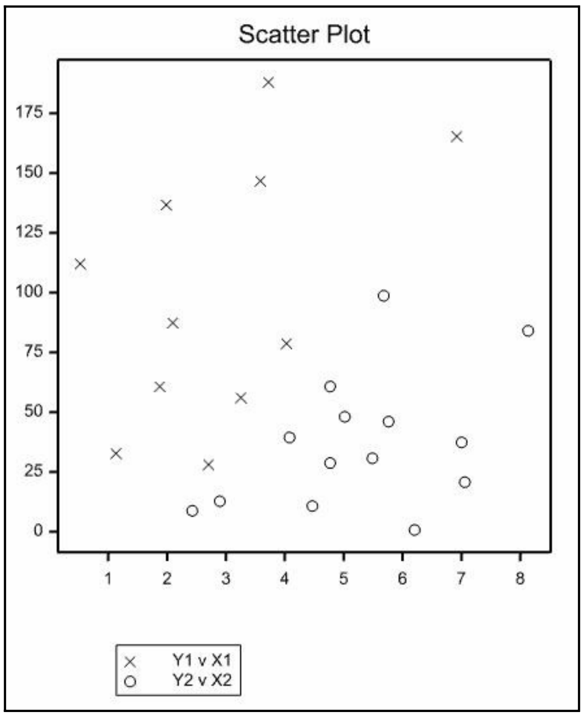

Figure 6.1a

In the simplest use of `DGRAPH`, you just need to specify the x- and y-coordinates of the points to be plotted and, if required, a title for the plot. For example, the statement:

```
DGRAPH [TITLE='Scatter
Plot'] Y1,Y2; X1,X2
```

generates the graph shown in Figure 6.1a. There are separate parameter lists for the y- and x-coordinates, which are processed in parallel so that the graph contains plots of `Y1` versus `X1` and `Y2` versus `X2`. The `TITLE` option provides a title for the graph, which is drawn at the top of the plot. It can be up to 80 characters in length, and must consist of one line of text only.

However, there are many more aspects of the output that can be controlled when producing a graph, and it is not feasible to allow all of these to be specified by the options or parameters of DGRAPH. The syntax would have become very complicated, and you would have had to specify all the relevant settings every time that DGRAPH was used. Instead additional directives are used to set up and modify a graphical *environment* which contains most of the information required when plotting. Each time DGRAPH is used, it accesses the relevant information from this environment in order to determine how to construct the graph. Thus, to make a simple modification to a graph, for example to change the colour of the plotted symbols, you need make only that specific change; any other information that you have supplied previously will remain in force. This section illustrates some of the settings that can be used to control or modify the appearance of graphical output. The complete description of the various elements of the environment and the directives that can be used to define them is in Section 6.9.

All the elements of graphical output, such as symbols, lines, axes, titles, labels, annotation and filled polygons are drawn by *pens*, which have associated definitions covering various attributes, like colours, symbol types and fonts. The pen also indicates the plotting method, that is, what kind of plot is to be drawn. For example, the following statements can be used to plot the data and the fitted line from a regression of Logpress on Boiltemp (see 2:3.1 for full details):

```
PEN 1,2; METHOD=line,point; SYMBOL=0,1; COLOUR='black'
DGRAPH [TITLE='Simple Linear Regression'] Fitted,Logpress;\
  Boiltemp; PEN=1,2
```

This means that *pen* 1 will be used to plot a line through the points specified by Fitted and Boiltemp, and *pen* 2 will be used to plot the points specified by Logpress and Boiltemp. The corresponding output is shown in Figure 6.1b.

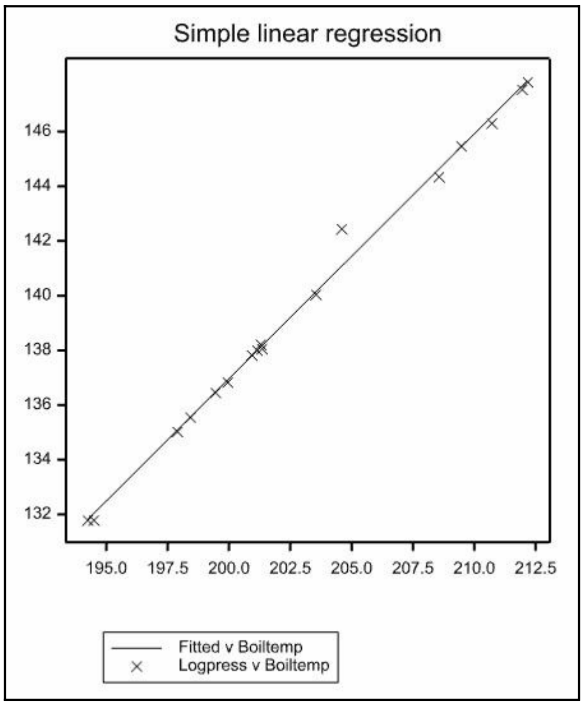

Figure 6.1b

Pens are available with numbers 1 up to 256, each with its own attribute settings, thus allowing a wide variety of styles within each plot. You can control which pens are used to plot the data, using the PEN parameter of DGRAPH as shown above. The PEN directive (6.9.8) can be used to define the various attributes of each pen, such as the colour, symbol type and line style (whether lines should be full, dotted or dashed). You can also specify labels to be plotted at each point, and control the size of symbols and text and the thickness of lines. When plotting a line you can switch off the symbols if you do not want to mark individual points, by setting SYMBOLS=0.

The XAXIS, YAXIS and ZAXIS directives (6.9.4, 6.9.5 and 6.9.6) allow you to specify the pens to be used for the axes and their titles, and the FRAME directive (6.9.3) enables you to specify the pen for the overall title. Unless you specify otherwise, Genstat uses pens with negative numbers for these features. Negatively numbered pens cannot be used for any other purposes, so this avoids any unintended side effects when pens are modified to change the main parts of the graph.

If the PEN parameter of DGRAPH is not specified, pen 1 is used for the first pair of Y and X structures, pen 2 for the second pair, and so on. So the pens need not have been specified explicitly in the `DGRAPH` statement above. The same convention applies to the pens used for different structures in histograms, pie charts and contour plots. The default settings for each pen are designed so that they will differ in appearance, for example by using different colours or line styles, depending on the output device. Thus, if you specify several data structures to appear in the same plot, the different sets of points or lines will be clearly distinguished by their different pens; see Figure 6.1a. The pens can be re-defined between `DGRAPH` statements in order to have a different effect each time.

You can also control the position and size of the graph. All graphical output is drawn in individual graphics *windows*. A window is a rectangular area of the screen. The position of the window is defined in terms of its lower and upper bounds in the vertical (y) and horizontal (x) directions using a data-independent coordinate system that ranges from 0.0 to 1.0 in each direction. You can use the default window positions defined by Genstat or you can use the `FRAME` directive (6.9.3) to define your own. The `WINDOW` option of `DGRAPH` indicates which window is to be used for the plot and `KEYWINDOW` specifies the location of the key.

The `FRAME` statement in lines 25 and 26 of Example 6.1 defines window 4 to have dimensions that will ensure that the y- and x-axes are suitably in proportion to show the shape of the series (see for example Cleveland & McGill 1987). The setting `KEYWINDOW=0` in `DGRAPH` in line 33 stops the key being displayed. The resulting graph is shown in Figure 6.1c.

Example 6.1

```
   2  " Wulfer's sunspot numbers, from Yule (1927)."
   3  READ    Sunspots

  Identifier   Minimum      Mean   Maximum    Values   Missing
    Sunspots    0.0000     44.76     154.4       176         0

  20  VARIATE Year; VALUES=!(1749...1924); DECIMALS=0
  21  " Set the window size so that the y- and x-axis length are in the
 -22    ratio 0.065:1, by calculating the upper y-bound to be
 -23    0.065 * x-axis length + y-margin (see 6.9.3)."
  24  CALCULATE Yup = 0.065*0.9 + 0.1
  25  FRAME   4; YLOWER=0.0; YUPPER=Yup; XLOWER=0.0; XUPPER=1.0;\
  26          YMLOWER=0.1; YMUPPER=0.0; XMLOWER=0.1; XMUPPER=0.0
  27  VARIATE Year; VALUES=!(1749...1924); DECIMALS=0
  28  " Set the window size so that the y- and x-axis length are in
 -29    the ratio 0.065:1, by calculating the upper y-bound to be
 -30    0.065 * x-axis length + y-margin (see 6.9.3)."
  31  CALCULATE Yup = 0.065*0.9 + 0.1
  32  FRAME   4; YLOWER=0.0; YUPPER=Yup; XLOWER=0.0; XUPPER=1.0;\
  33          YMLOWER=0.1; YMUPPER=0.0; XMLOWER=0.1; XMUPPER=0.0
  34  XAXIS   4; LOWER=1749; UPPER=1924; MARKS=!(1750,1800,1850,1900)
  35  YAXIS   4; LOWER=0; UPPER=175; MARKS=150
  36  PEN     1; METHOD=line; SYMBOL=0
  37  DGRAPH  [WINDOW=4; KEYWINDOW=0] Sunspots; Year
```

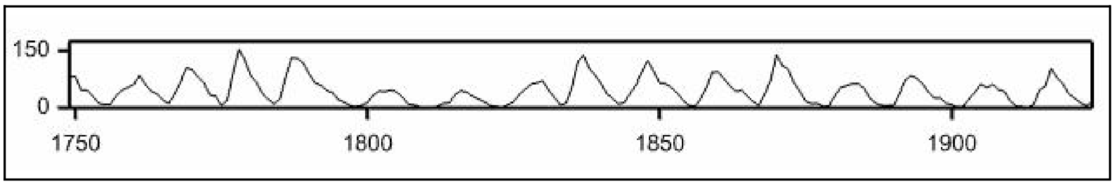

Figure 6.1c

Altogether, there are 256 windows, numbered from 1 up to 256. Windows are independent of one another and on most devices they are allowed to overlap or contain others. So it is possible to build up complex displays by a sequence of plotting commands. Details are given in Section 6.8.

For all the directives that produce graphics, the default window is window 1 and the default key window is window 2. The windows all have initial default sizes, as explained in 6.9.3. The default for window 2 is not very large. If you include too many variates in the plot, the key window may become full and a warning message will be printed. Also, very long identifier names or descriptions will be truncated if the width is insufficient. In either case you may want to use FRAME to increase the window size.

Each window has an associated definition for the axes that may be drawn in that window. The default definition will often be sufficient, but you can use the XAXIS, YAXIS and ZAXIS directives (6.9.4, 6.9.5 and 6.9.6) to control various aspects of the axes within each window, for example to add axis titles or to specify the spacing of tick marks or the position of labels. See lines 34 and 35 of Example 6.1, which define the lower and upper boundaries of the x- and y-axes, and the positions of the tick marks. These directives also allow you to specify which pens should be used for drawing the axes and adding annotation.

There are also ways in which you can control the output device. There are two options in DGRAPH (and in the other plotting directives) that can be used for this: SCREEN and ENDACTION. By default, when plotting a graph, Genstat will first clear the screen (or, equivalently, start a new page), but you can set option SCREEN=keep to preserve the current display. DGRAPH and D3GRAPH have an additional setting, SCREEN=resize, that will adjust the bounds of the axes, if necessary, to include the new information. Otherwise, the bounds are defined by the initial plot. DGRAPH can thus be used not only as a means of producing a self-contained picture, but also as a basic drawing tool to build up a complicated picture in several stages. Further details are given in Section 6.8

The ENDACTION option of DGRAPH is useful when producing graphs on an interactive graphical device; it specifies whether Genstat should pause at the completion of a graph, waiting for the user to press a key before continuing, or should immediately continue to the next statement. Where the screen has to switch between text and graphics modes, you would normally want to pause so that you could look at the graph; whereas using a windowed display, as on PCs running Windows, this is unnecessary unless several graphs are being drawn in succession, for example by a procedure. The default for ENDACTION, in the initial environment, uses the setting most suited to the current device. However, this can be modified by the DEVICE directive (6.9.1). ENDACTION is ignored when output is to a file.

6.2 High-resolution graphs in two and three dimensions

6.2.1 The DGRAPH directive

DGRAPH directive

Draws graphs on a plotter or graphics monitor.

Options

TITLE = *text*	General title; default *
WINDOW = *scalar*	Window number for the graphs; default 1
KEYWINDOW = *scalar*	Window number for the key (zero for no key); default 2
SCREEN = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (clear, keep, resize); default clea
KEYDESCRIPTION = *text*	Overall description for the key; default *
ENDACTION = *string token*	Action to be taken after completing the plot (continue, pause); default * uses the setting from the last DEVICE

	statement
HOTMENU = *matrices*	Defines sets of "hot" components for the user to select as shown or hidden by a menu in the Graphics Viewer
HOTCHOICE = *string token*	Whether one or several "hot" components can be displayed at a time (one, several); default seve

Parameters

Y = *identifiers*	Vertical coordinates
X = *identifiers*	Horizontal coordinates
PEN = *scalars*, *variates* or *factors*	Pen number for each graph (use of a variate or factor allows different pens to be defined for different sets of units); default * uses pens 1, 2, and so on for the successive graphs
DESCRIPTION = *texts*	Annotation for key
YLOWER = *identifiers*	Lower values for vertical bars
YUPPER = *identifiers*	Upper values for vertical bars
XLOWER = *identifiers*	Lower values for horizontal bars
XUPPER = *identifiers*	Upper values for horizontal bars
YBARPEN = *scalars*, *variates* or *factors*	Pens to use to draw the vertical bars; default -11
XBARPEN = *scalars*, *variates* or *factors*	Pens to use to draw the horizontal bars; -11
LAYER = *scalars*	"Layer" of the plot
UNITNUMBERS = *identifiers*	Specifies unit numbers to be used when points are selected in the graphics viewer; default * uses the actual unit numbers of the values in the X and Y structures
DISPLAY = *string tokens*	Whether to display each component initially in the graph (show, hide); default show
HOTCOMPONENT = *scalars*	Allows components of the graph (specified by pairs of Y and X settings) to be defined as "hot" components that can be shown or hidden through their association with "hot" points or using a menu in the Graphics Viewer
HOTDEFINITION = *matrices*	Define how to use points defined by the Y and X parameters as "hot" points in the Graphics Viewer to allow the user to decide whether other components of the graph are shown or hidden

The DGRAPH directive draws high-resolution graphs, containing points, lines or shaded polygons. The graph is produced on the current graphics device; this can be selected using the DEVICE directive as explained in 6.9.1. The WINDOW option defines the window, within the plotting area, in which the graph is drawn; by default this is window 1.

The Y and X parameters specify the coordinates of the points to be plotted; they must be numerical structures (scalars, variates, matrices or tables) of equal length. If any of the variates is restricted (4.4.1), only the subset of values specified by the restriction will be included in the graph. The restrictions are applied to the Y and X variates in pairs, and do not carry over to all the variates in a list. For example, suppose the variate Y1 is restricted but the variate Y2 is not. The statement

```
    DGRAPH Y1,Y2; X
```

will plot the subset of values of Y1 against X, but all the values of Y2 against X. Conversely, if X were restricted the subset would be plotted for both Y1 and Y2. Any associated structures, like variates specified by the PEN parameter or factors used to provide labels for the points, must be of the same length as Y and X.

Each pair of Y and X structures has an associated pen, specified by the PEN parameter. By default, pen 1 is used for the first pair, pen 2 for the second, and so on. The type of graph that is produced is determined by the METHOD setting of that pen. This can be point, to produce a point plot or scatterplot; line to join the points with straight lines; monotonic, open or closed to plot various types of curve through the points; spline to plot a smoothing spline fitted to the points; or fill to produce shaded polygons. In the initial graphics environment, all the pens are defined to produce point plots. This can be modified using the METHOD option of the PEN directive (6.9.8). Other attributes of the pen can be used to control the colour, symbols and labels as described in 6.9.8.

With METHOD=fill, the points defined by the Y and X variates are joined by straight lines to form one or more polygons which are then filled in the colour specified for the pen. The JOIN parameter of PEN determines the order in which the points are joined; with the default, ascending, the data are sorted into ascending order of x-values, while with JOIN=given they are left in their original order. There should be at least three points when using this method.

A warning message is printed if the data contain missing values. The effect of these depends on the type of graph being produced, as follows. If the method is point there will be no indication on the graph itself that any points were missing (but obviously none of the points with missing values for either the y- or x-coordinate can be included in the plot). If a line or curve is plotted through the points there will be a break wherever a missing value is found; that is, line segments will be omitted between points that are separated by missing values. When using METHOD=fill missing values will, in effect, define subsets of points, each of which will be shaded separately. Note, however, that the position of the missing values within the data will differ according to whether or not the data values have been sorted; this is controlled by the JOIN parameter of PEN, as described above. If the data are sorted, units with missing x-values are moved to the beginning.

The PEN parameter can also be set to a variate or factor, to allow different pens to be used for different subsets of the units. With a factor, the units with each level are plotted separately, using the pen defined by the ordinal number of the level concerned. If PEN is set to a variate, its values similarly define the pen for each unit. For example, if you fit separate regression lines to some grouped data, you can easily plot the fitted lines in just two statements, one to set up the pens and one to plot the data:

```
PEN 1...Ngroups; METHOD=line; SYMBOL=0
DGRAPH Fitted; X; PEN=Groups
```

By default, Genstat calculates bounds on the axes that are wide enough to include all the data; the range of the data is extended by five percent at each end, and the axes are drawn on the left-hand side and bottom edge of the graph. This can all be changed by the XAXIS and YAXIS directives (6.9.4 and 6.9.5), using the LOWER and UPPER parameters to set the bounds, and YORIGIN and XORIGIN to control the position of the axes. Other parameters allow you to control the axis labelling and style. If the axis bounds are too narrow, some points may be excluded from the graph, so that *clipping* occurs. If the plotting method is point, Genstat ignores points that are out of bounds. For other settings of METHOD, lines are drawn from points that are within bounds towards points that are out of bounds, terminating at the appropriate edge. Clipping may also occur if the method is monotonic, open or closed and you have left Genstat to set default axis bounds, because these methods fit curves that may extend beyond the boundaries. If this occurs, you should use the XAXIS and YAXIS directives to provide increased axis bounds. When you use several DGRAPH statements with SCREEN=keep to build up a complex graph, the axes are drawn only the first time, and the same axes bounds are then used for the subsequent graphs. You should then define axis limits that enclose all the subsequent data. Alternatively, if you set SCREEN=resize, the axes and their bounds will be adjusted, if necessary, to enclose the additional information. Axes are drawn only if SCREEN=clear, or the specified window has not been used since the screen was last cleared, or the window has been redefined by a FRAME statement.

DGRAPH allows *error bars* to be included in the plot. You might want to use these, for example, to show confidence limits on points that have been fitted by a regression (Part 2 Chapter 3). Error bars are requested by setting the YLOWER and YUPPER parameters to variates defining the lower and upper values for the error bar to be drawn at each point. For example, if you know the standard error for each point, you could calculate and plot the bounds as follows:

```
    CALCULATE Barlow = Y - 1.96 * Err
    & Barhigh = Y + 1.96 * Err
    DGRAPH Y; X; YLOWER=Barlow; YUPPER=Barhigh
```

(this would give a 95% confidence interval assuming that the y-values come from a Normal distribution). The error bar is drawn from the lower point to the upper point at the associated x-position; the bar will be drawn even if the corresponding y-value (or y-variate) is missing. If the lower value is missing, or the YLOWER parameter is not set, only the upper section of the bar is drawn; likewise if the upper value is missing only the lower section is drawn. Similarly, parameters XLOWER and XUPPER allow you to plot horizontal bars at each point.

The YBARPEN and XBARPEN parameters define the pens to be used for the vertical and horizontal bars, respectively, with the default to use pen −11. Similarly to the PEN parameter, they can be set to either scalars, factors or variates. For each group of units defined by the setting of PEN, DGRAPH will use the first pen that it finds for that group in the setting supplied by YBARPEN and XBARPEN. (So YBARPEN and XBARPEN cannot define more detailed groupings of the points than those defined by PEN.) For example:

```
    VARIATE [VALUES=1,1,2,2,3,3] Pvar
    &       [VALUES=4,4,5,5,6,6] Ybvar
    &       [VALUES=7,7,8,8,9,10] Xbvar
    DGRAPH  Y; X; PEN=Pvar; YLOWER=Ylow; YUPPER=Yupp;\
            XLOWER=Xlow; XUPPER=Xupp;\
            YBARPEN=Ybpen; XBARPEN=Xbpen
```

The first two points here will be plotted in pen 1 with vertical bar in pen 4 and horizontal bar in pen 7. The third and fourth points will be plotted in pen 2 with vertical bar in pen 5 and horizontal bar in pen 8. The fifth and sixth points will be plotted in pen 3 with vertical bar in pen 6 and horizontal bar in pen 9. Notice, that the horizontal bar for the sixth point will be plotted in pen 9 *not* pen 10, as it is in the same PEN group as the (earlier) fifth point which has pen 9 for the horizontal bar. However, if PEN is not set to a factor or variate, the YBARPEN and XBARPEN settings define the groups.

The KEYWINDOW option specifies the window in which the key appears; by default this is window 2. Alternatively, you can set KEYWINDOW=0 to suppress the key. The key contains a line of information for each pair of Y and X structures, written with the associated pen. This will indicate the symbol used, the line style (for a plotting method of line or curve) or a shaded block to illustrate the colour (when METHOD=fill), the name of the structure (if any) defined by the LABELS parameter of PEN, and a description indicating the identifiers of the data plotted (for example Residuals v Fitted). Alternatively, you can supply your own key, using the DESCRIPTION parameter, and you can specify a title for the key using the KEYDESCRIPTION option. If you draw several graphs using SCREEN=keep or SCREEN=resize and the same key window, each new set of information is appended to the existing key, until the window is full.

If you have set the PEN parameter to a variate or factor in order to plot independent subsets of the data, the key will contain information for each subset. If the LABELS parameter of PEN has been used to specify labels for the points, each line of the key will contain the label corresponding to the first value of the subset, rather than the identifier of the labels structure itself. In lines 22 and 23 of Example 6.2.1 the factor Animal is used to label the points according to the type of animal. Alternatively, in lines 24 and 25 the animals are distinguished with different plotting symbols, by using the factor to specify different pens for the different types of animal. The resulting plots are in Figure 6.2.1.

Example 6.2.1

```
   2  " Use of factors for labels and pens."
   3  FACTOR [LABELS=!T(zebra,giraffe)] Animal
   4  READ   Animal,Height,Weight; FREPRESENTATION=labels

    Identifier   Minimum      Mean   Maximum    Values   Missing
        Height    0.3080     4.384     9.228        14         0
        Weight     18.74     111.9     181.6        14         0

    Identifier    Values   Missing    Levels
        Animal        14         0         2

  19  XAXIS  5,6; LOWER=0; UPPER=240
  20  YAXIS  5,6; LOWER=0; UPPER=10.5
  21  FRAME  7,8; YLOWER=0.3; YUPPER=0.5; XLOWER=0.075,0.59
  22  PEN    1; LABELS=Animal; SIZE=0.8
  23  DGRAPH [WINDOW=5; KEYWINDOW=7] Height; Weight
  24  PEN    1,2; SYMBOLS=2,7; LABELS=*
  25  DGRAPH [WINDOW=6; KEYWINDOW=8; SCREEN=keep] Height; Weight; PEN=Animal
```

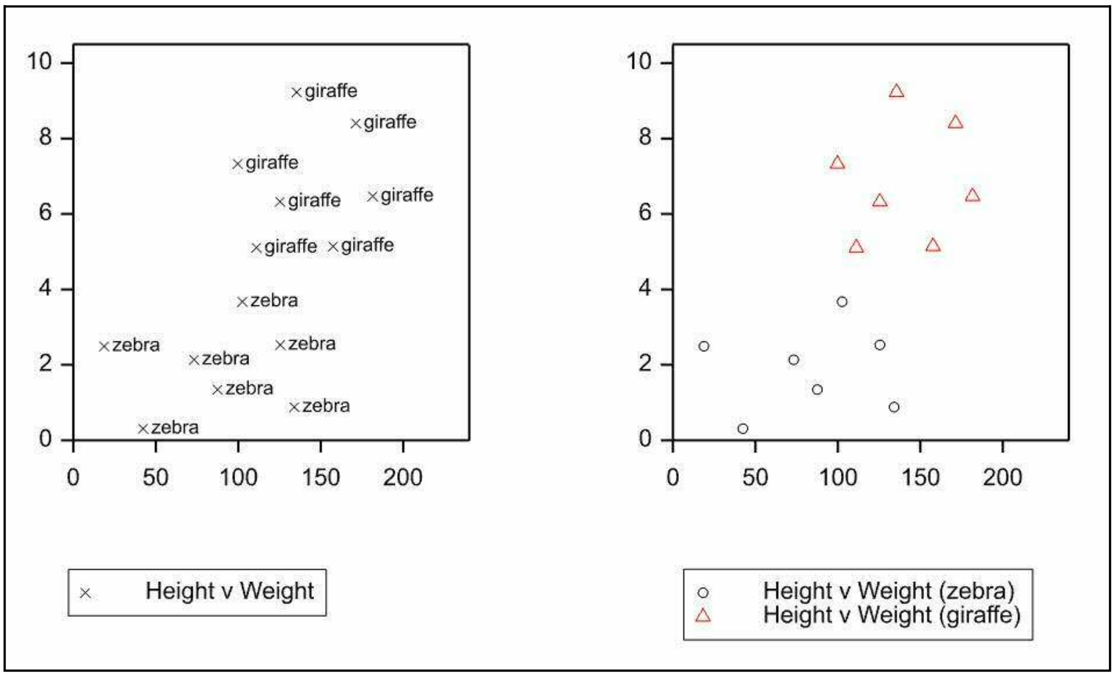

Figure 6.2.1

The `TITLE` option can be used to provide a title for the graph; this is plotted using pen -5. You can also put titles on the axes by using the `TITLE` parameter of the `XAXIS` and `YAXIS` directives.

The `SCREEN` option controls whether the graphical display is cleared before the graph is plotted, and the `ENDACTION` option controls whether Genstat pauses at the end of the plot, as described at the start of this section.

By default the sets of points defined by each pair of `Y` and `X` parameter settings are assumed to form separate, successive "layers" on the plot. So, if an area of the plot contains information (lines, symbols or labels) from several pairs of `Y` and `X` settings, the information from the later settings will overlay the information from earlier settings. You can control the orders of the layers by using the `LAYER` parameter to assign an explicit layer number to each pair of `Y` and `X` settings. The pairs of Y and X settings are then plotted in ascending order of layer numbers. These layer numbers also work across DGRAPH statements when you add to a plot by setting option SCREEN=keep or SCREEN=resize. So, for example, you can specify lower layer numbers to plot the new information "below" the layers formed by the earlier DGRAPH statement(s).

Usually all these components of the graph are shown when the graph is plotted. In Genstat *for Windows*, the Graphics Editor (which can be opened from the Edit menu on the menu bar of the Graphics Viewer) allows you to show or hide components, and the DISPLAY parameter of DGRAPH allows you to define whether a component should be shown or hidden in the initial graph displayed by the Graphics Viewer.

Alternatively, the Graphics Viewer itself can allow components to be shown or hidden, either by using their association with some "hot" points that have been defined on the graph, or by using a menu on its menu bar. These "hot" components are identified by defining a unique integer number for each one, using the HOTCOMPONENT parameter; if the component is not to be treated as "hot", HOTCOMPONENT should be left unset or given a missing value. Several pairs of Y and X parameter settings can be given the same number, so you can build up a "hot" component from more than one type of graphical item (e.g. from plotted points and shaded areas). "Hot" points are plotted within the graph using the Y, X and other parameters (e.g. PEN) in the usual way, as described above. The extra information, to define them as "hot", is supplied by setting the HOTDEFINITION parameter to a matrix with a row for each "hot" point, and a column for each type of "hot" component. The elements of the matrix specify the "hot" components to be associated with each "hot" point, using the numbers defined by the HOTCOMPONENT parameter. The menus in the Graphics Viewer can be made more informative, by defining textual labels for the rows and columns of the matrix (see the MATRIX directive); these are then used as annotation in the menus. Alternatively, if you set the HOTMENU option to a similar matrix, the Graphics Viewer will include a menu on its menu bar to allow users to choose whether "hot" components are shown or hidden. By default, users will be allowed to display several "hot" components at a time. However, you can set option HOTCHOICE=one to indicate that only one can be shown at a time. (The DISPLAY parameter should then be used to indicate which one, if any, should be shown on the initial graph.)

The Graphics Viewer also has a tool that allows you to select points, and copy their unit numbers onto the clipboard. Usually these numbers are simply the locations of the plotted values in the X and Y structures. However, you can use the UNITNUMBERS parameter to supply other numbers. (This may be useful if, for example, you are plotting sorted values.)

If you have a large number of points, it may be more effective to use a density plot. This displays the density of the points, in small regions of the x-y plane, as a surface plot. The procedure DXYDENSITY, that can be used, is described in Section 6.4.5.

6.2.2 The D3GRAPH directive

D3GRAPH directive

Plots a 3-dimensional graph.

Options

TITLE = *text*	General title; default *
WINDOW = *scalar*	Window number for the plots; default 1
KEYWINDOW = *scalar*	Window number for the key (zero for no key); default 2
ELEVATION = *scalar*	The elevation of the viewpoint relative to the surface; default 25 (degrees)
AZIMUTH = *scalar*	Rotation about the horizontal plane; the default of 225 degrees ensures that a point at the minimum x- and y-

	value is nearest to the viewpoint
DISTANCE = *scalar*	Distance of the viewpoint from the centre of the grid on the base plane; default * ensures that the data points fill the viewing area
SCREEN = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (clear, keep, resize); default clea
KEYDESCRIPTION = *text*	Overall description for the key; default *
ENDACTION = *string token*	Action to be taken after completing the plot (continue, pause); default * uses the setting from the last DEVICE statement

Parameters

X = *identifiers*	X-coordinates
Y = *identifiers*	Y-coordinates
Z = *identifiers*	Z-coordinates
PEN = *scalars*, *variates* or *factors*	Pen number for each graph (use of a variate or factor allows different pens to be defined for different sets of units); default * uses pens 1, 2, and so on for the successive graphs
DESCRIPTION = *texts*	Annotation for key
UNITNUMBERS = *identifiers*	Specifies unit numbers to be used when points are selected in the graphics viewer; default * uses the actual unit numbers of the values in the X and Y structures

The D3GRAPH directive produces high-resolution graphs, containing points, lines or filled shapes in three dimensions. The graph is produced on the current graphics device which can be selected using the DEVICE directive. The WINDOW option defines the window, within the plotting area, in which the graph is drawn; by default this is window 1.

The position of the viewpoint is specified in polar coordinates, using the options ELEVATION, DISTANCE and AZIMUTH. These define the angle of elevation, in degrees, above the base plane of the surface, distance from the centre of this plane, and angular position relative to the vertical z-axis, respectively.

The default settings of ELEVATION, DISTANCE and AZIMUTH have been chosen to produce a reasonable display of most situations; but if, for example, some parts of the plot are obscured they can be modified to obtain a better view. Altering the value of AZIMUTH will, in effect, rotate the plot in the horizontal plane about a vertical axis drawn through the centre of the plot; the default value of 225 degrees ensures that a point with the minimum x- and y-value would be at the corner nearest the viewpoint.

The X, Y and Z parameters specify the coordinates of the points to be plotted; they must be numerical structures (scalars, variates, factors, matrices or tables) of equal length. If any of the variates or factors is restricted, only the subset of values specified by the restriction will be included in the graph. The restrictions are applied to the X, Y and Z variates or factors in parallel sets, and do not carry over to other variates or factors in the list. Any associated structures, like variates specified by the PEN parameter or factors used to provide labels for the points, must be of the same length as X, Y and Z.

Each set of X, Y and Z structures has an associated pen, specified by the PEN parameter. By default, pen 1 is used for the first set, pen 2 for the second, and so on. The type of graph that is produced is determined by the METHOD setting of that pen. This can be point, to produce a point plot or scatterplot; line to join the points with straight lines; or fill to produce shaded objects. In the initial graphics environment, all the pens are defined to produce point plots. This can be modified using the METHOD option of the PEN directive (6.9.8). Other attributes of the pen can be used to control the colour, symbols and labels.

With METHOD=fill, the points defined by the X, Y and Z variates are joined by straight lines to form one or more polygons or polyhedrons which are then filled in the colour specified for the pen. The JOIN parameter of PEN is ignored for this directive. The points are plotted in the order in which they occur in the data.

A warning message is printed if the data contain missing values. The effect of these depends on the type of graph being produced, as follows. If the method is point there will be no indication on the graph itself that any points were missing (but obviously none of the points with missing values for either the x-, y- or z-coordinate can be included in the plot). If a line is plotted through the points there will be a break wherever a missing value is found; that is, line segments will be omitted between points that are separated by missing values. When using METHOD=fill missing values will, in effect, define subsets of points, each of which will be shaded separately.

The PEN parameter can also be set to a variate or factor, to allow different pens to be used for different subsets of the units. With a factor, the units with each level are plotted separately, using the pen defined by the level concerned. If PEN is set to a variate, its values similarly define the pen for each unit. For example, if you fit separate regression lines to some grouped data, you can easily plot the fitted lines in just two statements, one to set up the pens and one to plot the data:

```
PEN 1...Ngroups; METHOD=line; SYMBOL=0
D3GRAPH Fitted; X1; X2; PEN=Groups
```

By default, Genstat calculates bounds on the axes that are wide enough to include all the data; the range of the data is extended by five percent at each end, and the axes are drawn on the left-hand side and bottom edge of the graph. This can all be changed by the XAXIS, YAXIS and ZAXIS directives (6.9.4, 6.9.5 and 6.9.6) using the LOWER and UPPER parameters to set the bounds, and XORIGIN, YORIGIN and ZORIGIN to control the positions of the axes. Other parameters allow you to control the axis labelling and style. If the axis bounds are too narrow, some points may be excluded from the graph, so that *clipping* occurs. If the plotting method is point, Genstat ignores points that are out of bounds. For other settings of METHOD, lines are drawn from points that are within bounds towards points that are out of bounds, terminating at the appropriate edge. Clipping may also occur if the method is monotonic, open or closed and you have left Genstat to set default axis bounds, because these methods fit curves that may extend beyond the boundaries. If this occurs you should use the relevant axis directive to provide increased axis bounds. When you use several D3GRAPH statements with SCREEN=keep to build up a complex graph, the axes are drawn only the first time, and the same axes bounds are then used for the subsequent graphs. You should then define axis limits that enclose all the subsequent data. Alternatively, if you set SCREEN=resize, the axes and their bounds will be adjusted, if necessary, to enclose the additional information. Axes are drawn only if SCREEN=clear, or the specified window has not been used since the screen was last cleared, or the window has been redefined by a FRAME statement.

The KEYWINDOW option specifies the window in which the key appears; by default this is window 2. Alternatively, you can set KEYWINDOW=0 to suppress the key. The key contains a line of information for each pair of Y and X structures, written with the associated pen. This will indicate the symbol used, the line style (for a plotting method of line or curve) or a shaded block to illustrate the colour (when METHOD=fill), the name of the structure (if any) defined by the LABELS parameter of PEN, and a description indicating the identifiers of the data plotted (for example Residuals v Fitted). Alternatively, you can supply your own key, using the DESCRIPTION parameter, and you can specify a title for the key using the KEYDESCRIPTION option. If you draw several graphs using SCREEN=keep or SCREEN=resize and the same key window, each new set of information is appended to the existing key, until the window is full.

If you have set the PEN parameter to a variate or factor in order to plot independent subsets of the data, the key will contain information for each subset. If the LABELS parameter of PEN has been used to specify labels for the points, each line of the key will contain the label corresponding to the first value of the subset, rather than the identifier of the labels structure itself.

The Graphics Viewer has a tool that allows you to select points, and copy their unit numbers onto the clipboard. Usually these numbers are simply the locations of the plotted values in the `X` and `Y` structures. However, you can use the `UNITNUMBERS` parameter to supply other numbers. (This may be useful if, for example, you are plotting sorted values.)

The `TITLE` option can be used to provide a title for the graph. You can also put titles on the axes by using the `TITLE` parameters of the `XAXIS`, `YAXIS` and `ZAXIS` directives (6.9.4, 6.9.5 and 6.9.6). The `SCREEN` option controls whether the graphical display is cleared before the graph is plotted and the `ENDACTION` option controls whether Genstat pauses at the end of the plot.

Example 6.2.2 shows the use of `D3GRAPH` to plot three of the variables in Fisher's Iris data. The resulting graph is in Figure 6.2.2. The plot uses the sphere symbol, which is one of the symbols provided specially for three-dimensional plots (see 6.9.8). The axes are labelled. There is therefore no need for a key.

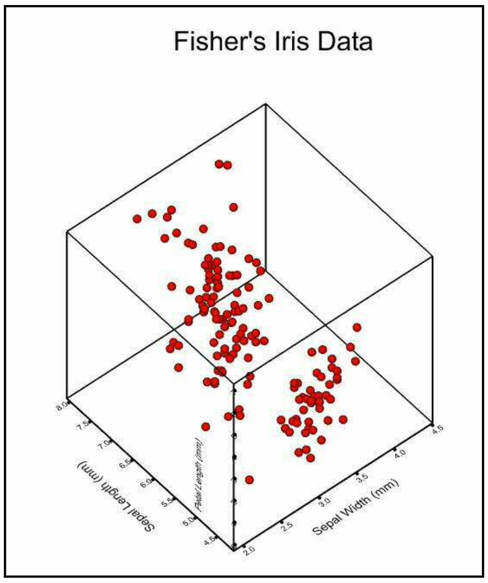

Figure 6.2.2

Example 6.2.2

```
   2  VARIATE [NVALUES=150] Plength,Pwidth,Slength,Swidth
   3  READ Slength,Swidth,Plength,Pwidth

    Identifier   Minimum      Mean   Maximum    Values   Missing
       Slength     4.300     5.843     7.900       150         0
        Swidth     2.000     3.057     4.400       150         0
       Plength     1.000     3.758     6.900       150         0
        Pwidth    0.1000     1.199     2.500       150         0

 154  FACTOR    [NVALUES=150; LABELS=!t(Setosa,Versicolor,Virginica); \
 155            VALUES=50(1,2,3)] Species
 156  PEN       1; SYMBOL='sphere'; COLOUR='red'; SIZE=2
 157  XAXIS     1; Title='Sepal Width (mm)'
 158  YAXIS     1; Title='Sepal Length (mm)'
 159  ZAXIS     1; Title='Petal Length (mm)'
 160  D3GRAPH   [TITLE='Fisher''s Iris Data'; AZIMUTH=230; ELEVATION=50; KEY=0]\
 161            Swidth; Slength; Plength
```

6.3 Histograms and bar charts

Histograms are used to represent the distribution of a set of data values. In the standard histogram, the range of the data is partitioned into consecutive intervals. The histogram has a bar for each interval, and the height of the bar represents the number of data values that the interval contains. Essentially, the histogram provides a graphical representation of a one-way table of counts; see Example 6.3.1a. The `DHISTOGRAM` directive (6.3.1) provides high-resolution plots of histograms with a single classifying factor. If you have two classifying factors, you can use the D3HISTOGRAM directive (6.4.4), which provides a plot in three dimensions: one for the lengths of the bars, and two for the classifying factors.

The bar chart differs from the histogram in that the factor classifying the table can represent any type of grouping, not simply a partitioning of a range of numerical values: for example a bar chart might present sales of a products in different areas, or sales of different types of product. Also, the table need not contain counts, but may contain any numerical values, for example profits (and losses), or yields of a crop. Bar charts with either one or two classifying factors can be plotted using the BARCHART directive (6.3.2).

6.3.1 The DHISTOGRAM directive

DHISTOGRAM directive

Draws histograms or bar charts on a plotter or graphics monitor.

TITLE = *text*	General title; default *
WINDOW = *scalar*	Window number for the histograms; default 1
KEYWINDOW = *scalar*	Window number for the key (zero for no key); default 2
LIMITS = *variate*	Variate of group limits for classifying DATA variates into groups; default *
LOWER = *scalar*	For a DATA variate, this specifies the lower limit of the first bar; default * takes the minimum value of the variate
UPPER = *scalar*	For a DATA variate, this specifies the upper limit of the last bar; default * takes the maximum value of the variate
NGROUPS = *scalar*	When LIMITS and BINWIDTH are not specified, this defines the number of groups into which a DATA variate is to be classified; default is then 10, or the integer value nearest to the square root of the number of values in the variate if that is smaller
BINWIDTH = *scalar*	When LIMITS is unset the range of a DATA variate is split into equal intervals known as "bins" to form the groups, this option can set the bin widths (alternative is to set the number of groups using NGROUPS)
FIXEDBARWIDTH = *string token*	Whether to plot the histogram with bars of equal width (no, yes); default no
BARCOVERING = *scalar*	What proportion of the space allocated along the x-axis each bar should occupy; default * gives proportion 1 for a DATA variate, and 0.8 for a factor or table (thus giving a gap between each bar)
BARSCALE = *scalar*	Width of bar for which one unit of bar length represents one unit of data; default * uses the width of the narrowest bar
LABELS = *text*	Group labels; default *
APPEND = *string token*	Whether or not the bars of the histograms are appended together (yes, no); default no
ORIENTATION = *string token*	Direction of the plot (horizontal, vertical); default vert
OUTLINE = *string token*	Where to draw outlines (bars, perimeter); default bars
PENOUTLINE = *scalar*	Pen to use for the outlines; default -8
SCREEN = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (clear, keep); default clea
KEYDESCRIPTION = *text*	Overall description for the key; default *

`ENDACTION` = *string token*	Action to be taken after completing the plot (`continue`, `pause`); default `*` uses the setting from the last `DEVICE` statement

Parameters

`DATA` = *identifiers*	Data for the histograms; these can be either a factor indicating the group to which each unit belongs, a variate whose values are to be grouped, or a one-way table giving the height of each bar
`NOBSERVATIONS` = *tables*	One-way table to save numbers in the groups
`GROUPS` = *factors*	Factor to save groups defined from a variate
`PEN` = *scalars* or *variates*	Pen number(s) for each histogram; default `*` uses pens 2, 3, and so on for the successive structures specified by `DATA`
`DESCRIPTION` = *texts*	Annotation for key

`DHISTOGRAM` plots high-resolution histograms or bar charts, depending on the input supplied by the `DATA` parameter. This can be either a list of variates, a list of factors or a list of one-way tables.

For a `DATA` variate, a histogram is produced. This summarizes the distribution of the variate by counting the number of values within a set of intervals defined by the `LIMITS`, `NGROUPS` or `BINWIDTH` options. The histogram contains a "bar" for each interval, with area proportional to the number of values found there.

Example 6.3.1a produces a histogram from a variate called `Data`, whose values are listed (in numerical order) in lines 4 and 5. The resulting graph is in Figure 6.3.1a.

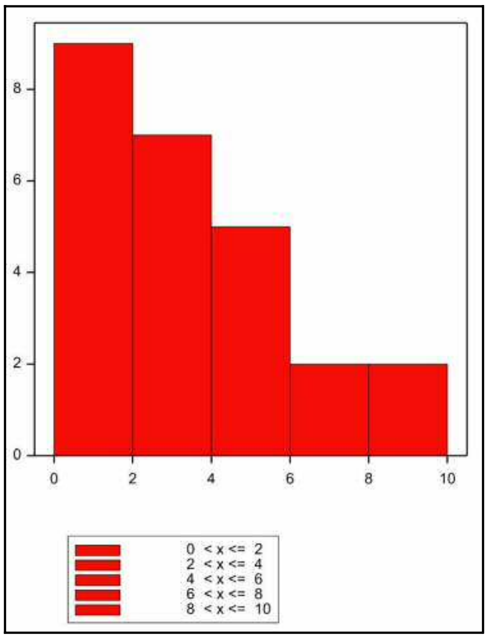

Figure 6.3.1a

Example 6.3.1a

```
  2   VARIATE Data
  3   READ    [PRINT=data,errors] Data
  4    0  1  1  1  1  2  2  2  2  3  3  4  4  4  4  4
  5    5  5  6  6  6  7  8  9  9  :
  6   DHISTOGRAM Data
```

You can define the boundaries between each interval using the `LIMITS` option. Alternatively, instead of setting `LIMITS`, you can specify the width of each interval using the `BINWIDTH` option. Or, instead of setting `LIMITS` or `BINWIDTH`, you can specify the number of groups using the `NGROUPS` option. Finally, if none of these options is set, Genstat defines the number of groups to be 10, or the integer value nearest to the square root of the number of values in the first `DATA` variate if that is smaller. The range of the histogram is specified by the `LOWER` and `UPPER` options. LOWER defines the lower limit of the first interval; by default this is set by making the width of the first bar equal to the width of the second bar, or it is the minimum value of the variates if that would otherwise be below the first bar. UPPER defines the upper limit of the last interval; by default this is set by making the width of the final bar equal to the width of the last-but-one bar, or it is the maximum value of the variates if that would otherwise be above the final bar. The bars are perpendicular to the x-axis, and this is labelled with the positions of the interval boundaries.

Bar charts are given if DATA is set to factors or tables. These differ from histograms in that there is no longer the concept of dividing the x-axis into a set of contiguous intervals. Instead we have a set of bars located at equal intervals along the x-axis. Figure 6.3.1b shows an example, generated by Example 6.3.1b below.

If DATA is set to a list of factors, the bars are labelled by the labels, if available, or otherwise the levels of the first factor. If DATA is set to a list of tables, the labelling is given by the levels/labels of the factor classifying the first table. A DATA table defines the heights of each bars directly (from the value in the corresponding cell of the table). With a factor, Genstat first constructs a table giving the replications of the factor levels. So the height of each bar is equal to the number of units of the factor with the corresponding level of the factor.

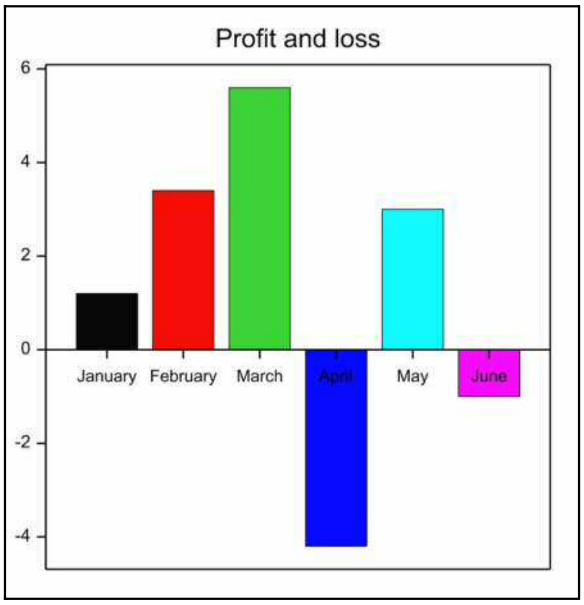

Figure 6.3.1b

Example 6.3.1b

```
2  FACTOR [LABELS=!t(January,February,March,April,May,June)] Month
3  TABLE  [CLASSIFICATION=Month; VALUES=1.2,3.4,5.6,-4.2,3.0,-1] Results
4  DHISTOGRAM [TITLE='Profit and loss'; WINDOW=1; KEYWINDOW=0] Results
```

The bars in a bar chart always have equal widths. With a histogram, the default is for the bar widths to be equal to the widths of the underlying intervals. However, you can request equal bar widths by setting option FIXEDBARWIDTH=yes. The BARCOVERING option indicates what proportion of the space allocated along the x-axis each bar should occupy. For a histogram the default is 1, while for bar charts it is 0.8 (thus giving a gap between each bar).

The BARSCALE option controls how the lengths of the bars correspond to units of data. The length of each bar is calculated as (data-value × BARSCALE)/bar-width. By default, BARSCALE is set to the width of the narrowest bar. So for that bar, the length will correspond directly to the data units.

The WINDOW option defines the window where the histogram is plotted, and the KEYWINDOW option similarly specifies where the key should appear. You can set either of these to zero if you want to suppress the corresponding output. Titles can be added to the histogram and key using the TITLE and KEYDESCRIPTION options respectively.

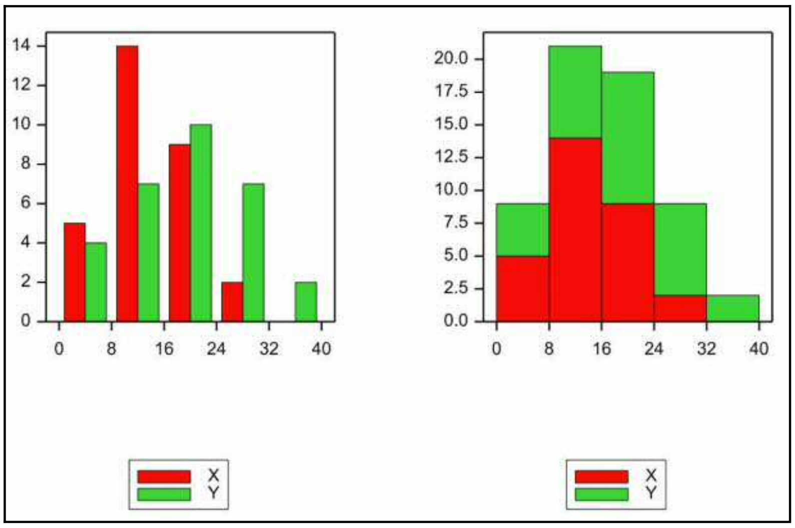

Figure 6.3.1c

The APPEND option controls the form of display used when the DATA parameter specifies a list of structures. These parallel histograms can be produced in one of two styles. By default (APPEND=no), the histogram contains a set of bars for each structure, drawn in parallel groups. This is used in line 15 of Example 6.3.1c to generate the histogram on the left-hand side of Figure 6.3.1c. Alternatively, if you set APPEND=yes, the bars for the structures are concatenated into a single bars for each group, as in line 16 of Example 6.3.1c and the right-hand side of Figure 6.3.1c. The bottom portion of each bar then corresponds to the first structure, and the top to the last structure.

Example 6.3.1c

```
   2  READ  X,Y

    Identifier   Minimum      Mean   Maximum    Values   Missing
             X     2.801     14.25     29.01        30         0
             Y     4.069     19.33     39.01        30         0

  11  FRAME 7,8; YUPPER=0.45; XLOWER=0.2,0.7
  12  XAXIS 5,6; MARKS=!(0,8...40)
  13  YAXIS 5,6; MARKS=!(0,2...14),!(0,2.5...20)
  14  DHISTOGRAM [WINDOW=5; KEYWINDOW=7; APPEND=no] X,Y
  15  DHISTOGRAM [WINDOW=6; KEYWINDOW=8; SCREEN=keep; APPEND=yes] X,Y
```

The ORIENTATION option controls whether the bars of the histogram are plotted vertically (the default) or horizontally. When ORIENTATION=horizontal, the horizontal axis is taken to be the y-axis, so the same XAXIS and YAXIS settings can be used however the histogram is oriented.

The bars for each structure are all shaded according to the pen or pens that have been specified for that structure, using the PEN parameter. You can set PEN to a scalar to define a single pen to be used for all the bars, or to a variate to define a different pen for each bar. If PEN is not set, Genstat uses the pens in turn, pen 2 for the first structure, pen 3 for the second structure, and so on, so that a different shading is used for each structure. The relevant aspects of the pens should be set in advance, if required, using the COLOUR parameter of the PEN directive (6.9.8). Generally, however, the default attributes of the pens will be satisfactory.

The OUTLINE option controls whether lines are drawn around the bars or around the perimeter of the histogram. These are drawn using the pen specified by the PENOUTLINE option (default -8). You can suppress all the outlines by setting OUTLINE=*.

The axes of the histogram are formed automatically from the data. By default, the upper bound of the y-axis is set to be five percent greater than the height of the longest bar. If any of the bars has a negative height the lower bound is adjusted in a similar way, otherwise it is set to zero. As already mentioned, when the histogram is formed from a variate, the x-axis markings are set to indicate the limits of each bar or set of bars; when the data are provided in a factor the factor labels or levels are used to label the histogram bars, and when the bar heights are provided directly in a table the classifying factor of the table is used. You can control the form of the axes by using the XAXIS and YAXIS directives (6.9.4 and 6.9.5) to set the required attributes before the DHISTOGRAM directive is used.

The WINDOW parameter of XAXIS and YAXIS should be set to the window in which the histogram is to be plotted (controlled by the WINDOW option of DHISTOGRAM). The TITLE, LOWER, UPPER, MARKS and LABELS parameters control annotation. The UPPER parameter of YAXIS is particularly useful when you are plotting a series of histograms; by setting UPPER to a value larger than any of the bars in any of the histograms, you can ensure that they are all plotted on the same scale.

The histogram key consists of the title, if set by KEYDESCRIPTION, followed by a legend for each structure plotted. This consists of a small rectangle that is drawn in the same colour as that used in the histogram, followed by the identifier name or the piece of text specified by the DESCRIPTION parameter.

The SCREEN option controls whether the graphical display is cleared before the histogram is plotted and the ENDACTION option controls whether Genstat pauses at the end of the plot.

6.3.2 The BARCHART directive

BARCHART directive

Plots bar charts in high-resolution graphics.

Options

TITLE = *text*	General title; default *
WINDOW = *scalar*	Window number for the histograms; default 1
KEYWINDOW = *scalar*	Window number for the key (zero for no key); default 2
BARWIDTH = *scalar*, *variate* or *table*	Width(s) of the bars; default * sets equal widths to fill the x-axis
BARCOVERING = *scalar*	What proportion of the space allocated along the x-axis each bar should occupy; default * gives proportion 1 for a DATA variate, and 0.8 for a factor or table (thus giving a gap between each bar)
LABELS = *text*	Labels for the bars or groups of bars; default *
APPEND = *string token*	Whether or not the bars of the histograms are appended together (yes, no); default no
ORIENTATION = *string token*	Direction of the plot (horizontal, vertical); default vert
YSCALING = *string token*	What scale to use to label the y-axis (absolute, proportion, percentage); default abso
OUTLINE = *string token*	Where to draw outlines (bars, perimeter); default bars
PENOUTLINE = *scalar*	Pen to use for the outlines; default -9
SCREEN = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (clear, keep); default clea
KEYDESCRIPTION = *text*	Overall description for the key; default *
ENDACTION = *string token*	Action to be taken after completing the plot (continue, pause); default * uses the setting from the last DEVICE statement

Parameters

DATA = *tables* or *variates*	Heights of the bars in each bar chart
ERRORBARS = *scalars*, *tables* or *variates*	Error bars to be plotted above the bars of each bar chart
LOWERERRORBARS = *scalars*, *tables* or *variates*	Heights of error bars plotted below the bars of each bar chart; if any of these is omitted, the corresponding setting of ERRORBARS is used as the default so that the error bars will have equal heights above and below the bars of the bar chart
GROUPS = *factors*	Which factor of a 2-way table to use as the groups factor; default uses the second classifying factor
PEN = *scalars*, *tables* or *variates*	Pen number(s) for each bar chart; default * uses pens 2, 3, and so on for the successive structures specified by DATA
PENERRORBARS = *scalars*, *tables* or *variates*	Pen number(s) for the error bars; default −11
DESCRIPTION = *texts*	Annotation for key

BARCHART plots high-resolution bar charts. You can plot a single bar chart by setting the DATA parameter to a one-way table or a variate defining the heights of the bars. To plot several bar charts on the same graph, you can set DATA to a list of one-way tables or variates. These must all contain the same number of values, and any tables must be classified by the same factor. Alternatively, you can set DATA to a two-way table. The GROUPS parameter then specifies which of the two classifying factors is to be treated as the "groups" factor (by default this is the second factor). BARCHART now plots a bar chart for every level of the GROUPS factors, with bars defined by the other classifying factor; see Example 6.3.2 and Figure 6.3.2.

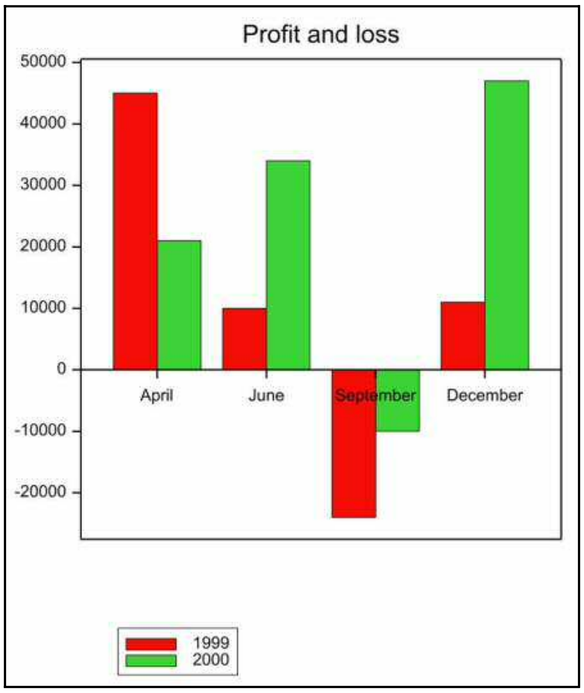

Figure 6.3.2

Labels can be supplied for the bars, using the LABELS option. If this is not set, the labels will be the labels or levels of the factor classifying the DATA tables, or the integers 1 upwards for a DATA variate.

By default, if there are several bar charts, they are plotted with their bars alongside each other. So BARCHART first plots the first bar of every bar chart, then the second bar, and so on. Alternatively, you can set option APPEND=yes to stack the bars into a single bar. The bottom portion of each bar then corresponds to the first bar chart, and the top to the last bar chart.

You can include error bars in a single bar chart or when several bar charts are plotted alongside each other, by specifying their heights with the ERRORBARS and LOWERERRORBARS parameters. The error bars take the form of a horizontal line joined by a vertical line of the specified height, above and below each bar. The ERRORBARS parameter specifies the heights of the error bars above the bars of the bar chart, and the LOWERERRORBARS parameter specifies the heights of the error bars below the bars. If LOWERERRORBARS is not specified, the error bars are assumed to have the same heights below and above the bars. You can set ERRORBARS and LOWERERRORBARS to a scalar if the heights are the same for every bar of a bar chart, or to a table or variate if different bars have error bars with different heights.

The ORIENTATION option controls whether the bars of the histogram are plotted vertically (the default) or horizontally. When ORIENTATION=horizontal, the horizontal axis is taken to be the y-axis, so the same XAXIS and YAXIS settings can be used however the histogram is oriented.

By default, Genstat uses pen 2 for the first bar chart, pen 3 for the second bar chart, and so on, so that a different colour is used for each one. Alternatively, you can define your own colours, using the PEN parameter. If you set PEN to a scalar, a single pen is used for all the bars. Alternatively, you can specify a variate or a table to define a different pen for each bar. The relevant aspects of the pens should be set in advance, if required, using the COLOUR parameter of the PEN directive (6.9.8). Generally, however, the default attributes of the pens will be satisfactory. Similarly, the PENERRORBARS parameter specifies the pen or pens to use for the error bars (default -11).

The bars in a bar chart usually have equal widths, defined to fill the available space along the x-axis. However, you can set your own widths by setting option BARWIDTH to either a scalar or a variate or table with as many values as the number of bars. The BARCOVERING option indicates what proportion of the space allocated along the x-axis each bar should occupy; the default is 0.8 (giving a gap between each bar).

The OUTLINE option controls whether lines are drawn around the bars or around the perimeter of the bar chart. These are drawn using the pen specified by the PENOUTLINE option (default -9). You can suppress all the outlines by setting OUTLINE=*.

The WINDOW option defines the window where the histogram is plotted, and the KEYWINDOW option similarly specifies where the key should appear. You can set either of these to zero if you want to suppress the corresponding output. Titles can be added to the histogram and key using the TITLE and KEYDESCRIPTION options respectively.

The SCREEN option controls whether the graphical display is cleared before the histogram is plotted and the ENDACTION option controls whether Genstat pauses at the end of the plot.

The axes of the plot are formed automatically from the data. By default, the upper bound of the y-axis is set to be five percent greater than the height of the longest bar. If any of the bars has a negative height the lower bound is adjusted in a similar way, otherwise it is set to zero. You can control the form of the axes by using the XAXIS and YAXIS directives to set the required attributes (such as titles) before the BARCHART directive is used. The YSCALING option controls the scale used to label the y-axis, with settings absolute, proportion or percentage; the default is absolute.

The key consists of the title, if set by KEYDESCRIPTION, followed information about each bar chart. You can specify a description for each bar chart using the DESCRIPTION parameter. If the DATA parameter was set to a list of one-way tables or variates, the default description takes the identifier of the table or variate. If DATA was set to a two-way table, the default descriptions are formed from the labels or levels of the GROUPS factor.

Example 6.3.2

```
FACTOR   [LEVELS=!(1999,2000)] Year
FACTOR   [LABELS=!t(April,June,September,December)] Month
TABLE    [CLASSIFICATION=Year,Month; VALUES=45000,10000,-24000,11000,\
         21000,34000,-10000,47000] Results
BARCHART [TITLE='Profit and loss'] Results
```

6.4 Plotting three-dimensional surfaces in high-resolution

The data for a three-dimensional surface consists of a grid of z-values or heights. The grid can be a rectangular matrix, a two-way table, or a pointer to a set of variates; the y-dimension is represented by the rows of the structure and the x-dimension by the columns. In each case there must be at least three rows and three columns of data (after allowing for any restrictions on a set of variates). Missing values are not permitted; that is, only complete grids can be displayed. If the grid is supplied as a table with margins, these will be ignored when plotting the surface.

Genstat provides four methods for plotting surfaces. A contour plot (`DCONTOUR`; 6.4.1) can be used to form a two-dimensional representation of the surface, in which contour lines are drawn to link points of equal height. The `DSHADE` directive (6.4.2) can produce a shade diagram. This is another two-dimension representation, in which each z-value is represented by a shaded rectangle indicating the value at that location, using its colour. This type of display is often used in a cluster analysis to display a similarity matrix, but it is also useful for the graphical display of spatial data. Alternatively, a three-dimensional representation of the surface, or *perspective view*, can be drawn using the `DSURFACE` directive (6.4.3), to display more fully the three-dimensional nature of the data. The grid can be viewed from any angle, allowing the investigation of features such as maxima, minima, valleys and plateaux. When the grid contains discrete data, a three-dimensional (or bivariate) histogram may be appropriate. This is produced using the `D3HISTOGRAM` directive (6.4.4), which forms the display by drawing cuboid blocks of the appropriate height at each (x,y) position.

6.4.1 The DCONTOUR directive

DCONTOUR directive

Draws contour plots on a plotter or graphics monitor.

Options

`TITLE` = *text*	General title; default `*`
`WINDOW` = *scalar*	Window number for the plots; default 1
`KEYWINDOW` = *scalar*	Window number for the key (zero for no key); default 2
`YORIENTATION` = *string token*	Y-axis orientation of the plot (`reverse`, `normal`); default `reve`
`ANNOTATION` = *string token*	How to annotate the contours (`levels`, `ordinals`); default `ordi` if there is a key, and `leve` if there is no key
`SCREEN` = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (`clear`, `keep`); default `clea`
`KEYDESCRIPTION` = *text*	Overall description for the key
`ENDACTION` = *string token*	Action to be taken after completing the plot (`continue, pause`); default `*` uses the setting from the last `DEVICE` statement

Parameters

`GRID` = *identifier*	Pointer (of variates representing the columns of a data matrix), matrix or two-way table specifying values on a regular grid
`PENCONTOUR` = *scalar*	Pen number to be used for the contours; default 1
`PENFILL` = *scalar* or *variate*	Pen number(s) defining how to fill the areas between contours, or 0 to leave the areas in the background colour; default 3
`PENHIGHLIGHT` = *scalar*	Pen number to use for highlighted contours; default 0

	i.e. no highlighting
`HIGHLIGHTFREQUENCY` = *scalar*	Frequency at which contours are to be highlighted; default 10
`NCONTOURS` = *scalar*	Number of contours; default 10
`CONTOURS` = *variate*	Positions of contours
`INTERVAL` = *scalar*	Interval between contours
`DESCRIPTION` = *text*	Annotation for key

The orientation of the y-axis of the contour plot is controlled by the `YORIENTATION` option. By default this is reversed, so that the element (1,1) of the grid is plotted at the top left-hand corner of the plot. The grid is thus in the same order as it would be if it were printed. This is convenient in Example 6.4.1a and Figure 6.4.1a, which shows a contour plot of ammonium nitrate concentrations in soil cores. The y-values represent depth in the soil, and so it is appropriate that they increase down the page.

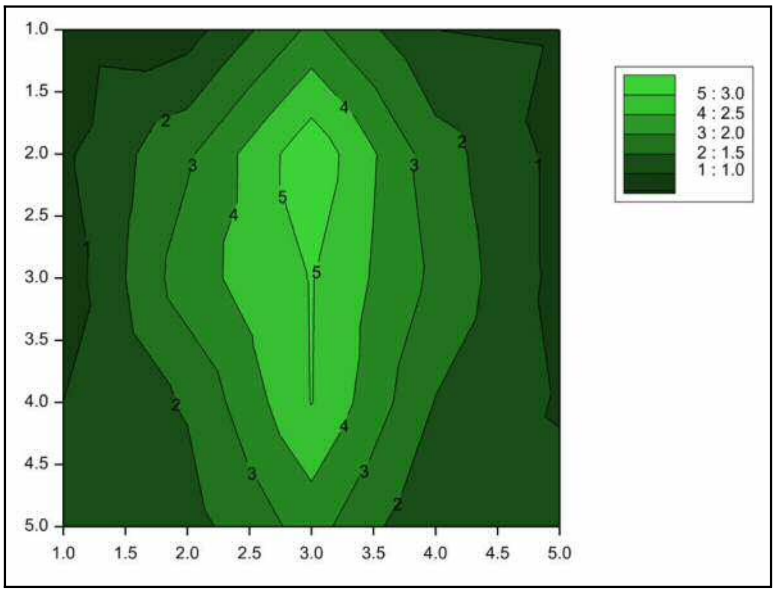

Figure 6.4.1a

Example 6.4.1a

```
" Core samples were taken from a wetland rice experiment to examine
  the leaching of ammonium nitrate. Three cores were taken at
  intervals of 5cm, and the concentration of ammonium nitrate was
  measured at depths of 4, 8, ... 20 cm. "
VARIATE   [NVALUES=5] Core[1...5]
READ      Core[]

 Identifier   Minimum      Mean   Maximum    Values   Missing
    Core[1]     5.000     8.200     11.00         5         0
    Core[2]     6.000     67.60     195.0         5         0
    Core[3]     129.0     940.6      2315         5         0
    Core[4]     10.00     36.00     77.00         5         0
    Core[5]     7.000     9.400     15.00         5         0

CALCULATE Core[] = LOG10(Core[])
FRAME     2; YLOWER=0.0; YUPPER=0.9; XLOWER=0.75; XUPPER=1.0
DCONTOUR  Core
```

Normally the data will lie on a regular grid but you can also specify an irregular grid as shown in line 10 of Example 6.4.1c; the `ROWS` and `COLUMNS` options of the `MATRIX` directive are set to variates containing the appropriate x- and y- values when the matrix `Function` is declared.

The `WINDOW` option defines the window where the contours are plotted, and the `KEYWINDOW` option similarly specifies where the key should appear. The grid axes are scaled so that the y- and x-dimensions (rows and columns respectively) will match the dimensions of the specified window: if you wish to preserve the "shape" of the grid you should use the `FRAME` directive

(6.9.3) to define a window whose y- and x-dimensions are in the same proportions as the grid dimensions, as shown in Example 6.4.1c. Titles can be added to these windows using the `TITLE` and `KEYDESCRIPTION` options. The `SCREEN` option controls whether the graphical display is cleared before the histogram is plotted and the `ENDACTION` option controls whether Genstat pauses at the end of the plot, as described in Section 6.1.

The heights of the contour lines are determined using the `NCONTOURS`, `CONTOURS` or `INTERVAL` parameters. The first possibility is to define the contours explicitly using the `CONTOURS` parameter. Alternatively, if `CONTOURS` is unset, `INTERVAL` can set the required interval between each contour. Or, if both `CONTOURS` and `INTERVAL` are unset, `NCONTOURS` defines the required number of lines. Genstat then partitions the range of data values accordingly to give `NCONTOURS` evenly-spaced contours (or fewer contours if there are insufficient distinct grid values).

The `ANNOTATION` option controls how the contours are labelled. The default is to label them by integers (`ordinals`) if there is a key, and by the actual heights (`levels`) if there is no key. Contour lines that are very short will not be labelled but their height can be determined from adjacent contours. Each line of the key occupies a space of height 0.02 (in normalized device coordinates; see 6.9.3), and the key window by default has room for a heading and nine contour levels. If necessary, the size of the window can be redefined using the `FRAME` directive.

The way in which the contour lines are drawn for each grid is determined by the pen that has been defined by the `PENCONTOUR` parameter of `DCONTOUR`; the default is to use pen 1. The relevant aspects of the pen should be set in advance, if required, using the `METHOD`, `COLOUR`, `LINESTYLE` and `THICKNESS` parameters of the `PEN` directive (6.9.8).

If the `PENCONTOUR` parameter is not used, the plotting method will be `line`, so that individual contours are made up of straight line segments. If curves are required, `METHOD` should be set to `monotonic` to use the method of Butland (1980), or `open` (or `closed`) to use the method of McConalogue (1970). Both these methods produce curves that are fitted to independent sets of interpolated points and can thus produce contour lines that cross, particularly if the supplied grid of data is coarse or in a region where the contour height is changing rapidly. If `METHOD` is set to other values, straight lines will be used to draw the contours.

The `PENHIGHLIGHT` parameter can specify a pen to use to highlight particular contours. The frequency of the highlighting is then determined by the `HIGHLIGHTFREQUENCY` parameter; by default every tenth contour is highlighted. This is illustrated in Example 6.4.1b and Figure 6.4.1b, where pen 2 is used to highlight every third contour.

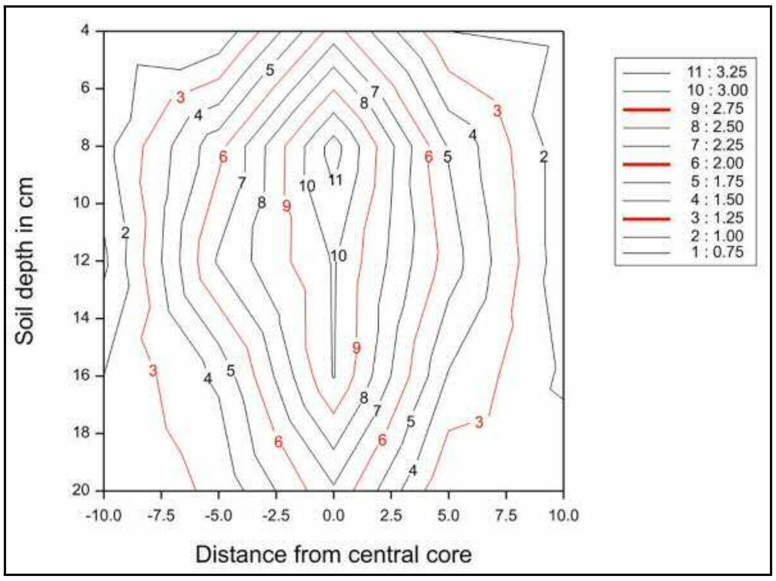

Figure 6.4.1b

Example 6.4.1b

```
XAXIS    1; TITLE='Distance from central core'; UPPER=10; LOWER=-10
YAXIS    1; TITLE='Soil depth in cm'; UPPER=4; LOWER=20
PEN      2; LINESTYLE=1; THICKNESS=3
DCONTOUR Core; PENCONTOUR=1; PENFILL=0; PENHIGHLIGHT=2; \
         HIGHLIGHTFREQUENCY=3; INTERVAL=0.25
```

The `PENFILL` parameter defines how to shade the areas between the contours. If `PENFILL` is set to zero, as in Example 6.4.1b, there is no shading i.e. the areas between the contours are left in the background colour. If `PENFILL` is set to a scalar, the shades are defined in increasing intensities of the colour of the specified pen. Alternatively, if `PENFILL` is set to a variate of length two, the pens are taken to define the shades at the minimum and maximum heights, and the other shades are interpolated between them. Finally, if `PENFILL` is set to a variate with more than two values, the shading uses the pens in the order in which they are given in the variate (recycling if insufficient pens are defined for the total number of contours). By default, `PENFILL=3`.

By default, on a colour device, the pens will be defined to use different colours, while on a monochrome device they will use different line styles. In line 18 of Example 6.4.1b, the `PEN` directive (6.9.8) specifies that pen 2 is to use a solid line style (like pen 1), and the `THICKNESS` is increased to produce the required highlighting.

By default, the axis bounds are determined from the grid. Normally the lower bound for each axis will be 1.0 and the upper bound will be the number of rows of the grid for the y-axis, and the number of columns for the x-axis. If a matrix is used to specify the grid, its row and column labels can be set to variates whose values will then be used to determine the axis bounds. The `XAXIS` and `YAXIS` directives (6.9.4 and 6.9.5) can be used to control how the axes are drawn (see Example 6.4.1b) or, by setting `STYLE=none`, to suppress them altogether.

In Example 6.4.1c, a matrix of function values is calculated over a regular range of y- and x-values, to produce the contour plot on the left-hand side of Figure 6.3.1c. The function is then recalculated on an irregular grid with the y- and x-values closest where the function is changing most rapidly, and the plot on the right-hand side is produced.

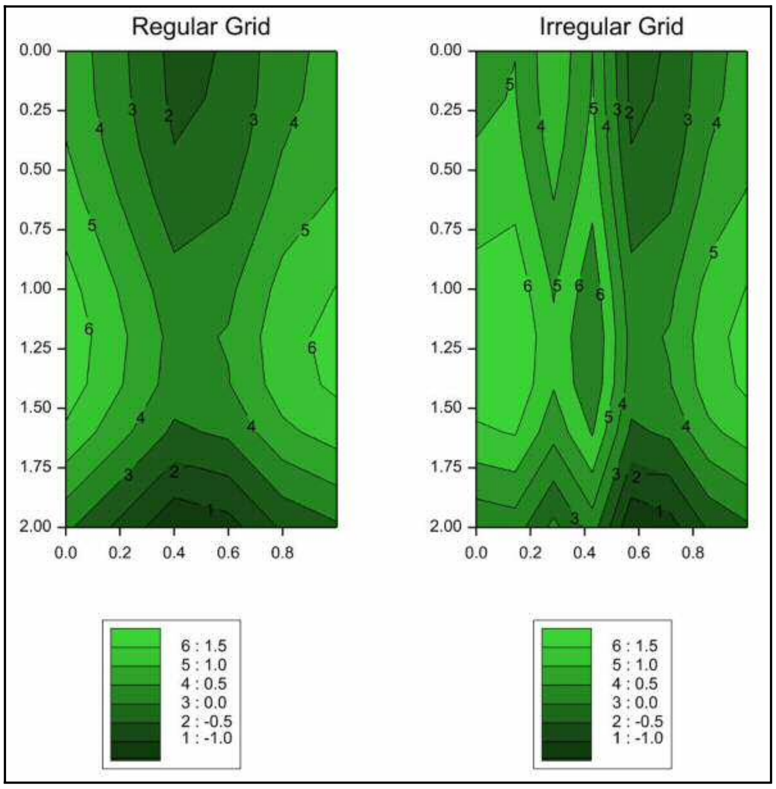

Figure 6.4.1c

Example 6.4.1c

```
VARIATE   Rows,Columns; VALUES=!(0.0,0.2...2.0),!(0.0,0.2...1.0)
CALCULATE Nrows,Ncolumns = NVALUES(Rows,Columns)
MATRIX    [ROWS=Rows; COLUMNS=Columns] X,Y,Function;\
          VALUES=!((#Columns)#Nrows),!(#Ncolumns(#Rows)),*
CALCULATE Function = COS(1/(X+0.1)**2) + SIN(Y**2)
FRAME     1; YLOWER=0.25; YUPPER=1; XLOWER=0; XUPPER=0.5
DCONTOUR  [TITLE='Regular Grid'] Function
VARIATE   Irregular; VALUES=!(0.0,0.1...0.4,0.6,0.8,1.0)
CALCULATE Nirregular = NVALUES(Irregular)
MATRIX    [ROWS=Rows; COLUMNS=Irregular] X,Y,Function;\
          VALUES=!((#Irregular)#Nrows),!(#Nirregular(#Rows)),*
CALCULATE Function = COS(1/(X+0.1)**2) + SIN(Y**2)
FRAME     3,4; YLOWER=0.25,0.0; YUPPER=1.0,0.25; XLOWER=0.5,0.675;\
          XUPPER=1
DCONTOUR  [TITLE='Irregular Grid'; WINDOW=3; KEYWINDOW=4; SCREEN=keep]\
          Function
```

6.4.2 The DSHADE directive

DSHADE directive

Plots a shade diagram of 3-dimensional data.

Options

TITLE = *text*	General title; default *
WINDOW = *scalar*	Window number for the graph; default 1
KEYWINDOW = *scalar*	Window number for the key (0 for no key); default 2
YORIENTATION = *string token*	Y-axis orientation of the plot (reverse, normal); default reve
GRIDMETHOD = *string token*	How to draw a grid around the elements of the matrix (present, complete); default pres
PENGRID = scalar	Pen to use for the grid; default -7
SCREEN = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (clear, keep); default clea
KEYDESCRIPTION = *text*	Overall description for the key
ENDACTION = *string token*	Action to be taken after completing the plot (continue, pause); default * uses the setting from the last DEVICE statement

Parameters

GRID = *symmetric matrix*, *matrix*, *table* or *pointer to variates*	Data to be plotted
PEN = *scalar* or *variate*	How to draw each shade
LIMITS = *variate*	Boundary values for changes in shade
NGROUPS = *scalar*	Number of groups to form from the data values (i.e. number of different shades)
INTERVAL = *scalar*	Interval between changes in shade
DESCRIPTION = *text*	Annotation for key

DSHADE produces a shaded representation of a rectangular or symmetric matrix using high-resolution graphics. Each element of the data matrix is represented by a shaded rectangle indicating the value at that location, using either colour or shading density. This type of display is often used in a cluster analysis to display a similarity matrix, but it is also useful for the graphical display of spatial data.

The data are specified by the GRID parameter, in either a matrix, a symmetric matrix (e.g. of similarities), a 2-way table or a pointer to a set of variates.

The range of data values corresponding to each shade are determined using the NGROUPS, the LIMITS or the INTERVAL parameter. The first possibility is to set LIMITS to a variate defining the boundaries on the data values where the shades change. Alternatively, if LIMITS is unset, NGROUPS can be used to define the required number of shades; Genstat then partitions the range of data values into that number of equal intervals (and shades each interval in a different way). Or, if both NGROUPS and LIMITS are unset, INTERVAL can set the interval between each change in shade. Finally, if none of these parameters is set, Genstat uses a different shade for each distinct data value. Missing values are ignored, thus leaving blank areas in the plot.

By default, the shades are drawn using pens 1, 2 onwards, with pen 1 being used for the lowest data values. Alternatively, you can specify the pen or pens explicitly, using the PEN parameter. If PEN is set to a scalar, the shades are defined in increasing intensities of the colour of the specified pen. Alternatively, if PEN is set to a variate of length two, the pens are taken to define the shades of the minimum and maximum data values, and the other shades are interpolated between them. Finally, you can set PEN to a variate with more than two values, and the shades use the pens in the order in which they are given in the variate (recycling if insufficient pens are defined for the total number of shades).

The shades are controlled by the current COLOUR setting of the pens. If the default settings do not produce a suitable display, these attributes should be set by a PEN statement before using DSHADE.

The GRIDMETHOD option specifies whether an outline should be drawn around each element of the matrix. The default setting, present, produces an outline for all values that are present; i.e. it ignores missing values. This is suitable where data have been sampled over an irregularly shaped area. Alternatively, with the complete setting, an outline is drawn around every element. Setting GRIDMETHOD=* stops the grid being drawn, which may be preferable if there are a large number of elements in the input data. The PENGRID option specifies which pen to use to draw the grid. The default is to use pen -7.

The YORIENTATION option controls the orientation of the y-axis. By default this is reversed, so that the data are in the same order as they would take if the data matrix were printed.

The TITLE, WINDOW, SCREEN and ENDACTION options are used to specify a title, the plotting window, whether the screen should be cleared first, and whether there should be a pause once the plotting is finished; as in other graphics directives. Similarly, the KEYWINDOW and KEYDESCRIPTION options and the DESCRIPTION parameters allow a key to be defined, if feasible for these plots with the current graphics device.

Example 6.4.2 uses `DSHADE` to display a similarity matrix for 16 types of Italian cars. (For details of how this matrix was formed see 2:6.1.2.) The resulting graph is in Figure 6.4.2.

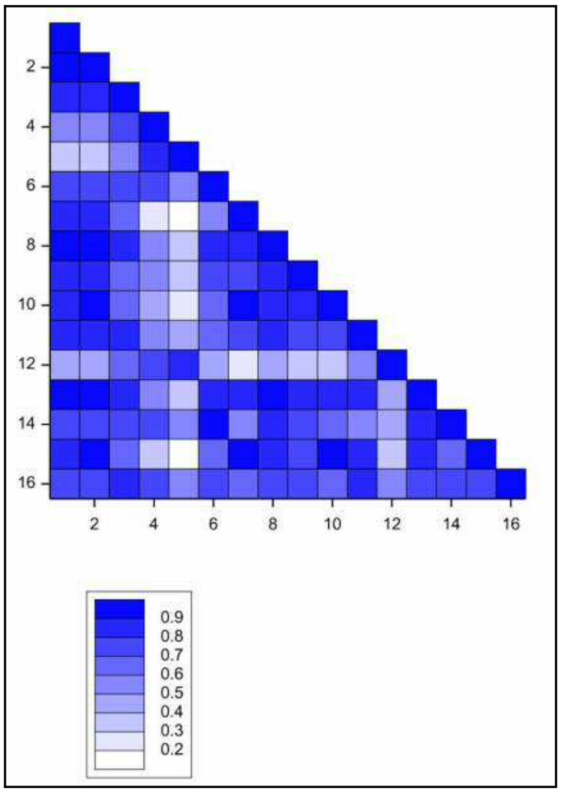

Figure 6.4.2

Example 6.4.2

```
SYMMETRIC [ROWS=!t(Estate,'Arna1.5','Alfa2.5',Mondialqc,\
          Testarossa,Croma,Panda,Regatta,Regattad,Uno,\
          X19,Contach,Delta,Thema,Y10,Spider)] Carsim
READ      Carsim

 Identifier   Minimum      Mean   Maximum    Values   Missing
     Carsim    0.1030    0.7249     1.000       136         0

FRAME  1; BOX=omit
PEN    9,10; COLOUR='white','blue'
DSHADE Carsim; INTERVAL=0.1; PEN=!(9,10)
```

6.4.3 The DSURFACE directive

DSURFACE directive

Produces perspective views of a two-way arrays of numbers.

Options

`TITLE` = *text*	General title; default *
`WINDOW` = *scalar*	Window number for the plots; default 1
`KEYWINDOW` = *scalar*	Window number for the key (zero for no key); default 2
`ELEVATION` = *scalar*	The elevation of the viewpoint relative to the surface; default 25 (degrees)
`AZIMUTH` = *scalar*	Rotation about the horizontal plane; the default of 225 degrees ensures that, with a square matrix `M`, the element

	M$[1;1] is nearest to the viewpoint
DISTANCE = *scalar*	Distance of the viewpoint from the centre of the grid on the base plane; default * gives a distance of 100 times the maximum of the x-range and the y-range
ZSCALE = *scalar*	defines the scaling of the z-axis relative to the horizontal (x-y) axes; default 1
SCREEN = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (clear, keep); default clea
KEYDESCRIPTION = *text*	Overall description for the key; default *
ENDACTION = *string token*	Action to be taken after completing the plot (continue, pause); default * uses the setting from the last DEVICE statement

Parameters

GRID = *identifier*	Pointer (of variates representing the columns of a data matrix), matrix or two-way table specifying values on a rectangular grid
PEN = *scalar*	Pen number to be used for the plot; default 1
PENFILL = *scalar* or *variate*	Pen number(s) defining how to fill the areas between contours (0 or * leaves the areas in the background colour); default 3
PENMESH = *scalar*	Pen number to use to draw the mesh (omitted if set to 0 or *); default 1
PENSIDE = *scalar*	Pen number to use to shade the sides of the surface (omitted if set to 0 or *); default *
NCONTOURS = *scalar*	Number of contours; default 10
CONTOURS = *variate*	Positions of contours
INTERVAL = *scalar*	Interval between contours
DESCRIPTION = *text*	Annotation for key

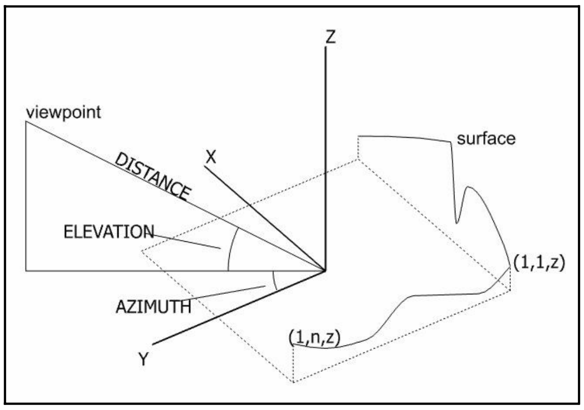

Figure 6.4.3a

The DSURFACE directive produces a perspective (or *conical*) projection of a surface, showing the view from a particular viewpoint. The position of this viewpoint is specified in polar coordinates, using the options ELEVATION, DISTANCE and AZIMUTH. These define the angle of elevation, in degrees, above the base plane of the surface, distance from the centre of this plane, and angular position relative to the vertical z-axis, respectively. This is illustrated in Figure 6.4.3a. The default settings of ELEVATION, DISTANCE and AZIMUTH have been chosen to produce a reasonable display of most surfaces; but if, for example, some parts of the surface are obscured by high points they can be modified to obtain a better view. Altering the value of AZIMUTH will, in effect, rotate the surface in the horizontal plane about a vertical axis drawn through the centre of the grid; the default value of 225 degrees ensures that the element in the first row and column of the grid is at the corner nearest the viewpoint. Small values of DISTANCE produce a perspective view; larger values, like the default of 100 times the maximum of the x-range and the y-range, effectively put the viewpoint at infinity to produce an "orthographic parallel projection".

The ZSCALE option specifies a scaling factor for the z-axis (or vertical axis) of the plotted surface. Generally values between 0.5 and 2.0 are most successful; large values result in a flatter surface, while smaller values produce a steep surface, accentuating changes in the data.

The TITLE, WINDOW, SCREEN and ENDACTION options are used to specify a title, the plotting window, whether the screen should be cleared first, and whether there should be a pause once the plotting is finished; as in other graphics directives. Similarly, the KEYWINDOW and KEYDESCRIPTION options and the DESCRIPTION parameters allow a key to be defined, if feasible for these plots with the current graphics device.

The PEN parameter specifies the pen to be used to plot the surface (by default, pen 1). The PEN directive can be used to modify the colour and the thickness of the pen, but the other attributes of the pen are ignored.

The NCONTOURS, CONTOURS and INTERVAL parameters control the contours drawn on the surface, if these are available on the current graphics device. The first possibility is to define the contours explicitly using the CONTOURS parameter. Alternatively, if CONTOURS is unset, INTERVAL can set the required interval between each contour. Or, if both CONTOURS and INTERVAL are unset, NCONTOURS defines the required number of lines. Genstat then partitions the range of data values accordingly to give NCONTOURS evenly-spaced contours (or fewer contours if there are insufficient distinct grid values).

The PENFILL parameter defines how to shade the areas between the contours. If this is set to a scalar, the shades are defined in increasing intensities of the colour of the specified pen. Alternatively, if PENFILL is set to a variate of length two, the pens are taken to define the shades at the minimum and maximum heights, and the other shades are interpolated between them. Finally, you can set PENFILL to a variate with more than two values, and the shading uses the pens in the order in which they are given in the variate (recycling if insufficient pens are defined for the total number of contours). The default is to use pen 3. However, if you set PENFILL to 0 or to a missing value, there will be no shading (that is, the areas between the contours will be in the background colour).

The PENMESH parameter specifies a pen to be used to draw a mesh on the surface. This consists of lines marking the points of the surface that lie above a rectangular grid on the xy plane. By default pen 1 is used, but if you set PENMESH to 0 or to a missing value the mesh is omitted.

The PENSIDE parameter defines the pen to use to shade the sides of the surface. There is no shading if this is set to 0 or a missing value, which is the default. The CFILL setting of the pen (see the PEN directive) specifies which colour is used.

Simple axes are drawn to indicate the directions in which x and y increase. The TITLE parameter of the XAXIS and YAXIS directives (6.9.4 and 6.9.5) can be used to add further annotation, as shown in lines 12 and 13 of Example 6.4.3 which produced the plots in Figures 6.4.3b and 6.4.3c. You can also use the UPPER parameter of ZAXIS (6.9.6) to truncate the grid, and the LOWER parameter to set the value for the base of the surface (line 16).

Example 6.4.3

```
   2  VARIATE   [VALUES=0.0,0.05...1.0] Values
   3  CALCULATE Nvalues = NVALUES(Values)
   4  MATRIX    [ROWS=Nvalues; COLUMNS=Nvalues] X,Y,Grid;\
   5            VALUES=!((#Values)#Nvalues),!(#Nvalues(#Values)),*
   6  CALCULATE Fx = EXP(-0.5*((X-0.3)/0.07)**2) \
   7                + 0.5*EXP(-0.5*((X-0.7)/0.12)**2)
   8  &         Fy = EXP(-0.5*((Y-0.3)/0.07)**2) \
   9                + 0.5*EXP(-0.5*((Y-0.7)/0.12)**2)
  10  &         [PRINT=summary] Grid = Fx*Fy+0.1

    Identifier   Minimum      Mean   Maximum    Values   Missing
```

```
        Grid    0.1000    0.1960     1.104       441         0    Skew

PEN       11; SIZE=4; COLOUR='black'
XAXIS     3; TITLE='The X axis'; PENTITLE=11
YAXIS     3; TITLE='The Y axis'; PENTITLE=11
ZAXIS     3; PENTITLE=11
DSURFACE  [WINDOW=3; KEY=0; TITLE='Default option settings'] Grid
ZAXIS     3; LOWER=0; UPPER=0.6
DSURFACE  [WINDOW=3; KEY=0; TITLE='Changes of viewpoint and z-axis';\
          AZIMUTH=120] Grid
```

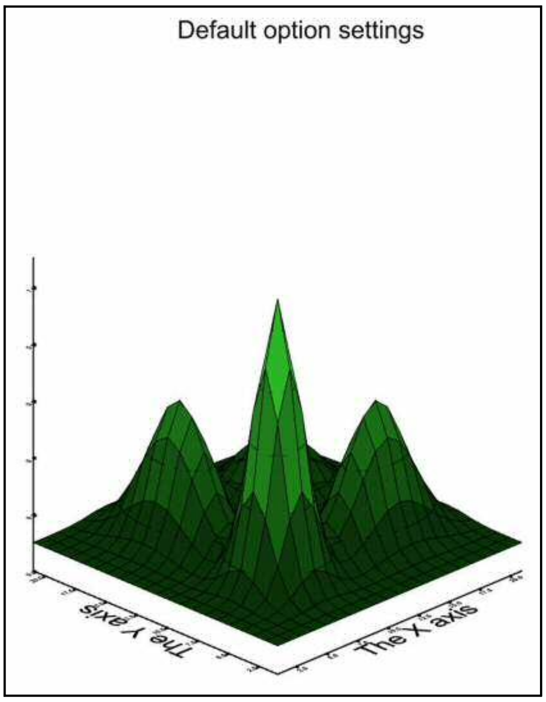

Figure 6.4.3a

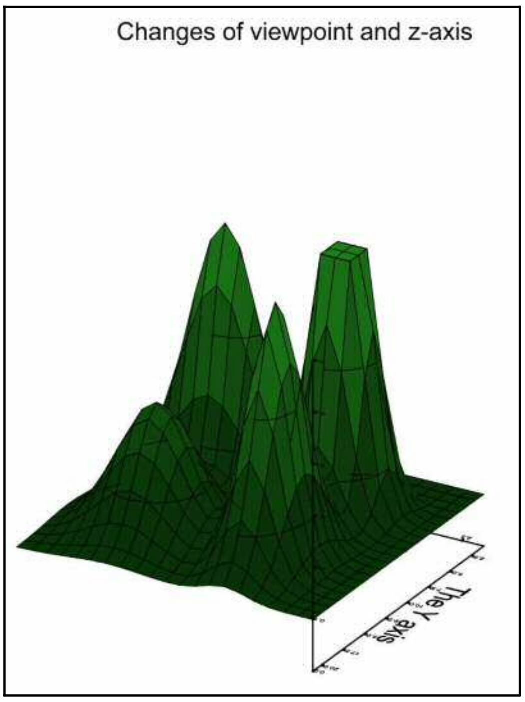

Figure 6.4.3a

6.4.4 Three-dimensional histograms: the `D3HISTOGRAM` directive

`D3HISTOGRAM` directive

Plots three-dimensional histograms.

Options

`TITLE` = *text*	General title; default `*`
`WINDOW` = *scalar*	Window number for the plots; default 1
`KEYWINDOW` = *scalar*	Window number for the key (zero for no key); default 2
`ELEVATION` = *scalar*	The elevation of the viewpoint relative to the surface; default 25 (degrees)
`AZIMUTH` = *scalar*	Rotation about the horizontal plane; the default of 225 degrees ensures that, with a square matrix `M`, the element `M$[1;1]` is nearest to the viewpoint
`DISTANCE` = *scalar*	Distance of the viewpoint from the centre of the grid on the base plane; default `*` gives a distance of 100 times the maximum of the x-range and the y-range
`SCREEN` = *string token*	Whether to clear the screen before plotting or to

	continue plotting on the old screen (`clear`, `keep`); default `clea`
`KEYDESCRIPTION` = *text*	Overall description for the key; default `*`
`ENDACTION` = *string token*	Action to be taken after completing the plot (`continue`, `pause`); default `*` uses the setting from the last `DEVICE` statement

Parameters

`GRID` = *identifier*	Pointer (of variates representing the columns of a data matrix), matrix or two-way table specifying values on a regular grid
`PEN` = *scalar*	Pen number to be used for the plot; default 3
`DESCRIPTION` = *texts*	Annotation for key

The preceding subsection described how the `DSURFACE` directive can be used to produce a perspective view of a surface. `D3HISTOGRAM` provides an alternative way of displaying such data, which may be more appropriate for example if the grid contains counts.

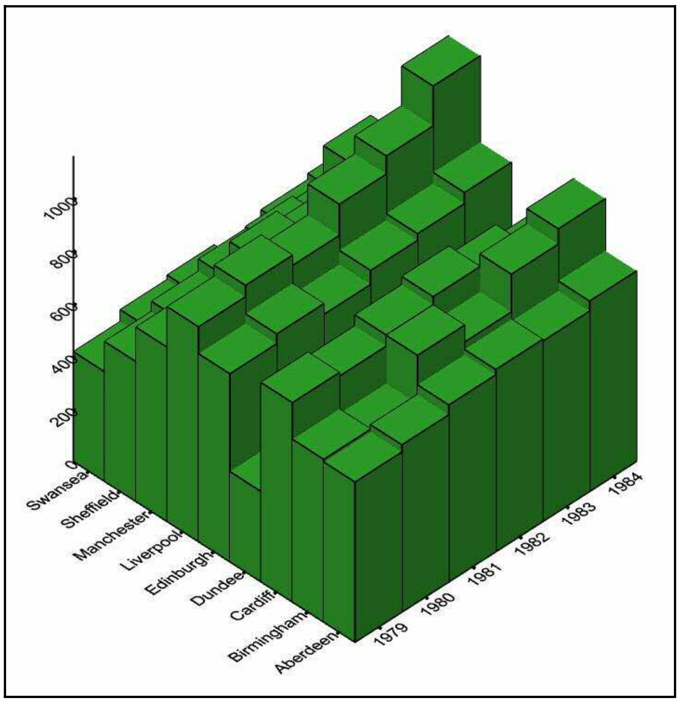

Figure 6.4.4

The position of the point from which the histogram is viewed is specified in polar coordinates, using the options `ELEVATION`, `DISTANCE` and `AZIMUTH` as with `DSURFACE` (6.4.3). The `TITLE`, `WINDOW`, `SCREEN` and `ENDACTION` options operate in the usual way, to specify a title, the plotting window, whether the screen should be cleared first, and whether there should be a pause once the plotting is finished. Similarly, the `KEYWINDOW` and `KEYDESCRIPTION` options and the `DESCRIPTION` parameters allow a key to be defined, if feasible for these plots with the current graphics device. The `PEN` parameter specifies the pen to be used to plot the histogram (by default, pen 3). The `PEN` directive (6.9.8) can be used to modify the colour and the thickness of the pen, but the other attributes of the pen are ignored.

Example 6.4.4 illustrates the use of `D3HISTOGRAM` by displaying the table `Sales` formed in Example 4.11.4; the resulting graph is in Figure 6.4.4. The `AZIMUTH` and `ELEVATION` options are used to obtain a clearer view of the surface, and `LOWER` option of the `ZAXIS` directive (6.9.6) is used to set the minimum z-value to zero. Note that when the grid is not square, as in this example, the y- and x-axes are scaled appropriately. This is also the case when using `DSURFACE`.

The axis labelling is derived from the grid, using the classifying factors if it is a table or the row and column labels if it is a matrix. Alternative labels can be supplied using the `LABELS` parameters of the `XAXIS` and `YAXIS` directives (6.9.4 and 6.9.5). If axis labels are not available, either from the grid or from an `XAXIS` or `YAXIS` statement, plain axes will be drawn in the style used by `DSURFACE`; these can be labelled using the `TITLE` parameter of `XAXIS` and `YAXIS`.

Example 6.4.4

```
" Plot 3-d histogram."
PEN   11; SIZE=2; COLOUR='black'
XAXIS 3; PENLABELS=11
YAXIS 3; MARKS=!(1...9); LABELS=Townname; PENLABELS=11
ZAXIS 3; LOWER=0; PENLABELS=11
D3HISTOGRAM [WINDOW=3; KEY=0; ELEVATION=40] Sales
```

6.4.5 Density plots: the DXYDENSITY procedure

DXYDENSITY procedure

Produces density plots for large data sets (D. B. Baird).

Options

PLOT = *string tokens*	How to plot the density (pointplot, shadeplot, contourplot, histogram, surface); default poin
NGROUPS = *scalar*	Number of sections into which to divide each axis (4-400); default 50
METHOD = *string token*	Method to use to smooth the density (thinplate, radialspline, tensorspline, kernel); default * i.e. none
DF = *scalar*	Degrees of freedom for smoothing methods (2-50); default 12
BANDWIDTH = *scalar*	Bandwidth for kernel smoothing (0-1); default 0.2
MEANFIT = *string tokens*	What smooth regression fits to the means to plot (yx, xy); default * i.e. none
NCONTOURS = *scalar*	Number of contours in the contour plot; default 9
SYMBOL = *string token*	Symbol to use in a point plot (circle, square); default circ
COLOURS = *text*, *variate* or *scalar*	Colour to use to draw the symbols, shades, contours or surface; default !(red, blue, black)
XTRANSFORM = *string token*	Transformed scale for the x-axis (identity, log, log10, logit, probit, cloglog, square, exp, exp10, ilogit, iprobit, icloglog, root); default iden
YTRANSFORM = *string token*	Transformed scale for the y-axis (identity, log, log10, logit, probit, cloglog, square, exp, exp10, ilogit, iprobit, icloglog, root); default iden
ZTRANSFORM = *string token*	Transformed scale for the z-axis (identity, percentile, root); default iden
WINDOW = *scalar*	Window number for the graphs; default 3
SCREEN = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (clear, keep, resize); default clea

Parameters

Y = *variate* or *factor*	Y-coordinates of the data
X = *variate* or *factor*	X-coordinates of the data

`TITLE` = *text*	Title for graph; default uses the names of the data and type of plot

Procedure `DXYDENSITY` produces a density plot of two variables, using high-resolution graphics. A density plot provides a better visual representation of the 2-dimensional spread of points than a scatter plot if there are a large number of points or many points overlap each other, and is quicker to plot. `DXYDENSITY` calculates the density of points in small regions of the x-y plane, and displays it as a surface plot.

The x and y axes are divided into equally spaced sections, to give a grid of rectangular cells covering the x-y plane. The density is calculated as the number of points that falls into each cell. The number of sections is specified by the `NGROUPS` option, as a scalar if the same number is required in each direction, or as a variate with two values to specify different numbers for the y-axis (first value) and the x-axis (second value). Having a large number of cells preserves more detail, but increases the time required to create and plot the graph.

The x- or y-axes can be transformed before forming the sections and calculating the density, by using the `XTRANSFORM` or `YTRANSFORM` options. The settings are the same as those of the `TRANSFORM` option of the `XAXIS` and `YAXIS` directives (see 6.9.4).

The `PLOT` option controls how the density is plotted, with settings:

`pointplot`	point plot , using the symbol size to indicate the number of points in each cell;
`shadeplot`	shade plot, using intensity of colour to indicate the number of points in each cell;
`contourplot`	contour plot, with contours showing the density;
`surface`	surface plot, with density as height;
`histogram`	3-dimensional histogram of the density.

By default `PLOT=pointplot`.

The density can be smoothed by using the `METHOD` option, with settings:

`thinplate`	a 2-dimensional thin plate spline is fitted to the counts using the `THINPLATE` procedure;
`radialspline`	a 2-dimensional radial spline is fitted to the counts using the `RADIALSPLINE` procedure;
`tensorspline`	a 2-dimensional tensor spline is fitted to the counts using the `TENSORSPLINE` procedure;
`kernel`	a 2-dimensional kernel smoother is fitted to the counts.

By default no smoothing is done.

The `DF` option specifies the number of degrees of freedom for the splines (default 12); smaller values make the surface smoother, and larger values allow it to be rougher. The `BANDWIDTH` option specifies the band width for kernel smoothing; larger values make the surface smoother, and smaller values allow it to be rougher.

The shape of each point in a point plot is specified by the `SYMBOL` option, as either a circle (default) or square. The `COLOURS` option specifies the colours that are used, in a scalar or a text or variate with up to three values. For a line plot, the first value specifies the colour for the points, and the second and third values define the colours for any lines fitted by the `MEANFIT` option. For a histogram, the first value of `COLOURS` defines the colour of the bars. For shade, contour and surface plot, if `COLOURS` has two or more values, the first is used for high densities, the second is used for low densities, and intermediate densities are plotted in the corresponding intermediate colour; if `COLOURS` has only one value, the low densities are plotted in white. If `COLOURS` has three values, the third is used for the contours of contour and surface plots.

The scaling of densities is controlled by the `ZTRANSFORM` option with settings:

`identity`	no scaling (default),
`root`	takes the square root of the densities, giving more

	emphasis to low counts,
`percentile`	takes a rank transform and plots these, so that percentiles are equally spaced.

The `MEANFIT` option allows you can to add a smoothing spline regression, of y on x or of x on y, to a point plot. The available settings are

`yx`	for a regression of y on x, and
`xy`	for a regression of x on y.

The `DF` option again specifies the number of degrees of freedom for the spline (default 12). By default neither are done.

The `Y` and `X` parameters specify the y- and x-coordinates of the data values, in either variates or factors. Their identifiers are used for the titles of the axes at the lower and left-hand edges of the graphics frame (i.e. page). You can also use the `TITLE` parameter to supply an overall title for the plot.

The `WINDOW` options specifies the number of the window to use for the plot, and the `SCREEN` option controls whether the screen is cleared first, as usual.

Figure 6.4.5 contains some density plots of log-ratio and intensity from GenePix microarray slides, plotted by the commands in Example 6.4.5. Further analyses can be found in the guide to the *Analysis of Microarray Data*, which can be accessed in Genstat *for Windows* from the Help menu on the menu bar.

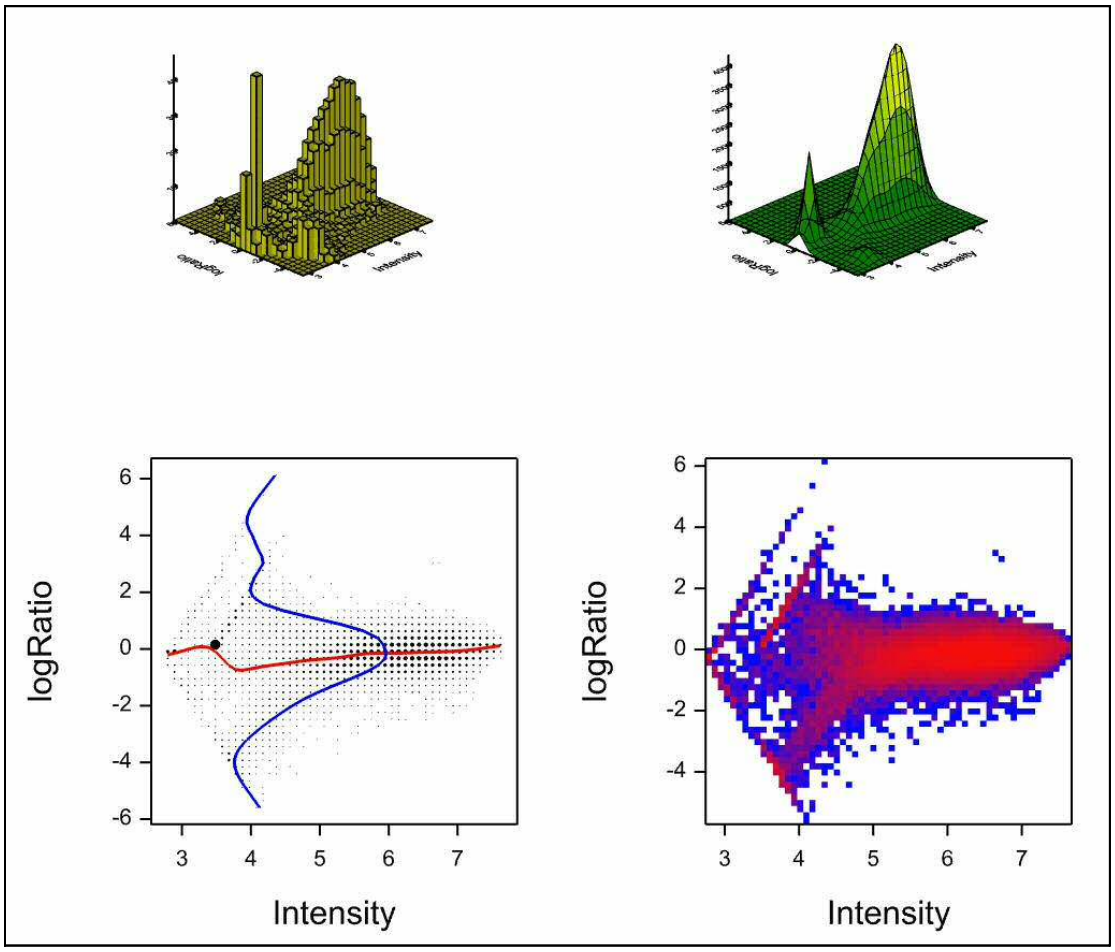

Figure 6.4.5

Example 6.4.5

```
" density plots of log-ratio and intensity from microarray slides "
SPLOAD     [PRINT=*] '%GENDIR%/Data/Microarrays/Data13-6-9.gwb'
RESTRICT   logRatio,Intensity; Intensity > 1
" histogram of square-root transformed densities "
DXYDENSITY [PLOT=histogram; ZTRANSFORM=root; NGROUPS=25;\
           COLOUR='yellow'; WINDOW=5]\
           Y=logRatio; X=Intensity; TITLE=''
" surface plot of kernel-smoothed densities "
DXYDENSITY [PLOT=surface; METHOD=kernel; BANDWIDTH=0.1;\
           WINDOW=6; SCREEN=keep; NGROUPS=25;\
           COLOUR=!t(yellow,green,black)] Y=logRatio; X=Intensity; TITLE=''
" point plot with regression splines of y on x and x on y "
DXYDENSITY [PLOT=POINT; MEANFIT=xy,yx; COLOUR=!t(black,red,blue);\
           WINDOW=7; SCREEN=keep] Y=logRatio; X=Intensity; TITLE=''
" shade plot "
DXYDENSITY [PLOT=SHADE; NGROUPS=60; ZTRANSFORM=percentile;\
           COLOUR=!t(red,blue); WINDOW=8; SCREEN=keep]\
           Y=logRatio; X=Intensity; TITLE=''
```

6.5 Displaying pictures

6.5.1 The `DBITMAP` directive

`DBITMAP` directive

Plots a bit map of RGB colours.

Options

`TITLE` = *text*	General title; default `*`
`WINDOW` = *scalar*	Window number for the graph; default 1
`YORIENTATION` = *string token*	Y-axis orientation of the plot (`reverse`, `normal`); default `reve`
`GRIDMETHOD` = *string token*	How to draw a grid around the elements of the matrix (`present`, `complete`); default `*` i.e. none
`PENGRID` = scalar	Pen to use for the grid; default `-7`
`SCREEN` = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (`clear`, `keep`); default `clea`
`ENDACTION` = *string token*	Action to be taken after completing the plot (`continue`, `pause`); default `*` uses the setting from the last `DEVICE` statement

Parameters

`BITMAP` = *symmetric matrix*, *matrix*, *table*, *pointer to variates* or *variate*	Data to be plotted
`ROWS` = *variate*	Row indexes for a `BITMAP` variate
`COLUMNS` = *variate*	Column indexes for a `BITMAP` variate

DBITMAP displays a picture, represented as a 2-dimensional bit map of RGB colours (see 6.8.9). The data are specified by the BITMAP parameter. Data values in a regular two-way grid can be specified by supplying their RGB colours in either a matrix, a symmetric matrix, a 2-way table or a pointer to a set of variates. Alternatively, you can specify irregular data by setting BITMAP to a variate of colours, and the ROWS and COLUMNS parameters to variates defining their row and column indexes. In Genstat *for Windows* you can form the bit map from an image file (JPG, GIF, TIF or PNG) using the IMPORT procedure, as shown in Example 6.5.1 and Figure 6.5.1.

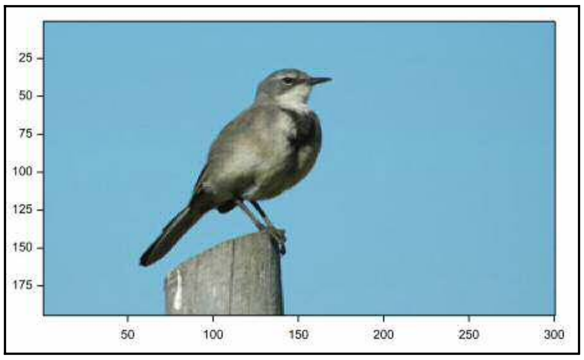

Figure 6.5.1

Example 6.5.1

```
IMPORT    [PRINT=*; RGBMETHOD=matrix] 'CapeWagtail.jpg'; COLUMNS='RGB'
" resize the window to match the dimensions of the bit map "
CALCULATE Nr = NROWS(RGB)
&         Nc = NCOLUMNS(RGB)
IF Nr < Nc
  FRAME   3; YUPPER=Nr/Nc
ELSE
  FRAME   3; XUPPER=Nc/Nr
ENDIF
DBITMAP   [WINDOW=3] RGB
```

The GRIDMETHOD option allows you to draw an outline around each element of the plot. The present setting produces an outline for all values that are present; i.e. it ignores missing values. This is suitable where data have been sampled over an irregularly shaped area. Alternatively, with the complete setting, an outline is drawn around every element. By default, no grid is drawn. The PENGRID option specifies which pen to use to draw the grid. The default is to use pen -7.

The YORIENTATION option controls the orientation of the y-axis. By default this is reversed, so that the data are in the same order as they would take if the data matrix were printed.

The TITLE, WINDOW, SCREEN and ENDACTION options are used to specify a title, the plotting window, whether the screen should be cleared first, and whether there should be a pause once the plotting is finished; as in other graphics directives.

6.6 Pie charts

6.6.1 The DPIE directive

DPIE directive

Draws a pie chart on a plotter or graphics monitor.

Options

TITLE = *text*	General title; default *
WINDOW = *scalar*	Window number for the pie chart; default 1
KEYWINDOW = *scalar*	Window number for the key (zero for no key); default 2
ANNOTATION = *string token*	Whether to annotate the slices by their percentages

	(`percentages`); default `perc`
`OUTLINE` = *string token*	Where to draw outlines (`slices`, `perimeter`); default `slices`
`PENOUTLINE` = *scalar*	Pen to use for the outlines; default -10
`SCREEN` = *string token*	Whether to clear the screen before plotting or to continue plotting on the old screen (`clear`, `keep`); default `clea`
`KEYDESCRIPTION` = *text*	Overall description for the key
`ENDACTION` = *string token*	Action to be taken after completing the plot (`continue`, `pause`); default `*` uses the setting from the last `DEVICE` statement

Parameters

`SLICE` = *scalars*	Amounts in each of the slices (or categories)
`PEN` = *scalars*	Pen number for each slice; default `*` uses pens 1, 2, and so on for the successive slices
`DESCRIPTION` = *texts*	Description of each slice

A pie chart is formed by taking the values of the scalars in the `SLICE` parameter, in order, and representing them by segments of a circle starting at "three o'clock" and working in an anti-clockwise direction. The angle subtended by each segment (and thus the area of the segment) is proportional to the value of the corresponding scalar. The values may be raw data or can be expressed as percentages (by ensuring they total to 100).

The colour used for each segment can be controlled using the `PEN` parameter. By default, pen 1 is used for the first segment, pen 2 for the second segment, and so on. The default colours differ from pen to pen, and can be modified using the `PEN` directive, as described in 6.9.8.

Lines 3 and 4 of Example 6.6.1 plot a pie chart with four slices, as shown in the top half of Figure 6.6.1.

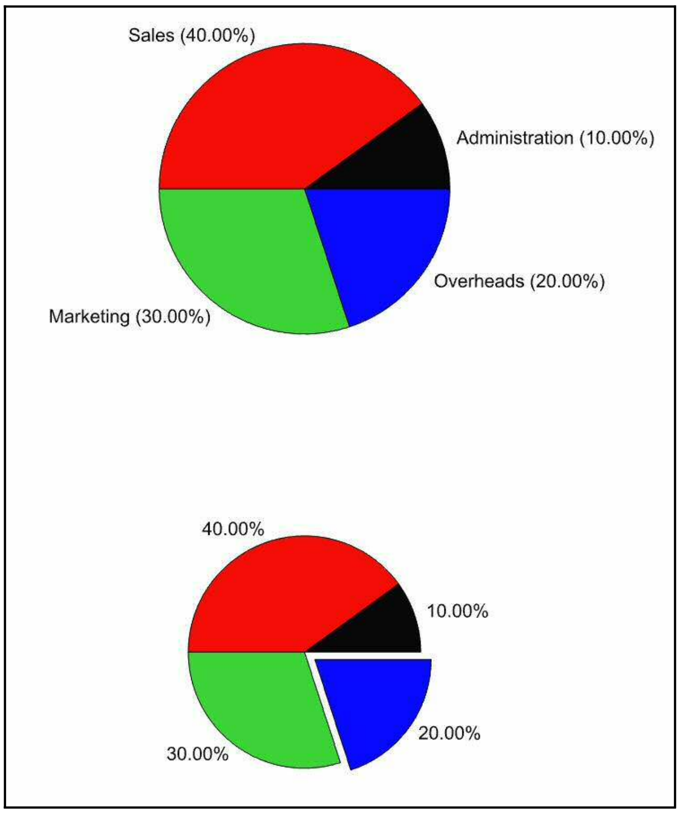

Figure 6.6.1

Example 6.6.1

```
  2  FRAME 1,2; YLOWER=0.5,0.0; YUPPER=1.0,0.5; XLOWER=0.0; XUPPER=1.0
  3  DPIE  [WINDOW=1; KEYWINDOW=0] 24.7,98.8,74.1,49.4; \
  4        DESCRIPTION='Administration','Sales','Marketing','Overheads'
  5  DPIE  [WINDOW=2; KEYWINDOW=0; SCREEN=keep;\
  6        ANNOTATION=percentage] 10,40,30,-20; \
  7        DESCRIPTION='Administration','Sales','Marketing','Overheads'
```

Individual segments can be displaced outwards from the centre, to obtain an "exploded" pie chart, as in the bottom half of Figure 6.6.1. The chosen segments are indicated by setting the corresponding scalars in the `SLICE` parameter list to negative values (see line 6 of Example 6.6.1).

The `WINDOW` and `KEYWINDOW` options specify the windows in which the pie chart and key are to be displayed. The shape of the pie chart is determined by the dimensions of the window; if it is not square the resulting pie chart will be elliptical.

Titles can be added using the `TITLE` and `KEYDESCRIPTION` options. The key produced for the pie chart is similar to that produced by the `DHISTOGRAM` directive. A shaded block is drawn for each segment, followed by the identifier name or the piece of text specified by the `DESCRIPTION` parameter. The key usually also gives the percentage contained by each slice, but you can suppress this by setting option `ANNOTATION=*`.

The `OUTLINE` option controls whether lines are drawn around the slices or around the perimeter of the pie chart. These are drawn using the pen specified by the `PENOUTLINE` option (default −10). You can suppress all the outlines by setting `OUTLINE=*`.

The `SCREEN` option controls whether the graphical display is cleared before the histogram is plotted and the `ENDACTION` option controls whether Genstat pauses at the end of the plot, as described at the start of this section.

6.7 Adding lines, annotation, error bars and customized keys to a graph

6.7.1 The `DTEXT` procedure

`DTEXT` procedure

Adds text to a graph (S.A. Harding).

Option

`WINDOW` = *scalar*	Window number of the graph; default 1

Parameters

`Y` = *variates* or *scalars*	Vertical coordinates
`X` = *variates* or *scalars*	Horizontal coordinates
`TEXT` = *texts*	Text to plot
`PEN` = *scalars*, *variates* or *factors*	Pens to use; default 1

The `DTEXT` procedure provides a convenient way of adding textual annotation or description to a plot. The text to plot is specified by the `TEXT` parameter. This can be either a single string, or a Genstat text structure containing several lines of text. The `Y` and `X` parameters specify where to plot the text, with scalars for a single string or line, or with variates for several lines. The `PEN` parameter specifies the pen or pens to use (default 1), and the `WINDOW` option specifies the window where the plot is taking place (default 1).

Example 6.7.1 uses `DTEXT` to insert the annotation onto the Venn diagram shown in Figure 6.7.1.

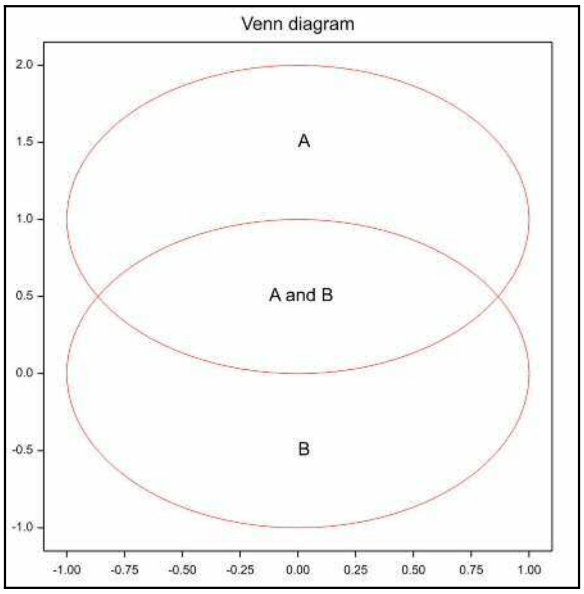

Figure 6.7.1

Example 6.7.1

```
VARIATE   [VALUES=0...100] theta
CALCULATE theta = 2 * C('pi') * theta / 100
&         X  = COS(theta)
&         Y1 = SIN(theta)
&         Y2 = Y1 + 1
PEN       2; METHOD=closed; SYMBOL=0; JOIN=given
DGRAPH    [TITLE='Venn diagram'; WINDOW=3; KEY=0] Y1,Y2; X; PEN=2
PEN       1; SIZE=1.5
DTEXT     [WINDOW=3] Y=1.5,0.5,-0.5; X=0,-0.125,0;\
          TEXT='A','A and B','B'; PEN=1
```

`DTEXT` adds the annotation to a particular plot. Alternatively, if you want to put annotation onto the full graphics frame, outside any of the plots, you can use the `DFRTEXT` procedure, which has very similar parameters.

6.7.2 The DREFERENCELINE procedure

DREFERENCELINE procedure

Adds reference lines to a graph (R.W. Payne).

Options

`ORIENTATION` = *string token*	Direction of the line (`horizontal`, `vertical`); default `hori`
`WINDOW` = *scalar*	Window in which to draw the line; default 1

Parameters

`POSITION` = *scalars*	Positions of the lines
`PEN` = *scalars*	Pen to use for each line
`LABEL` = *texts*	Text to plot alongside each line
`YLPOSITION` = *string tokens*	Position of the label in the y-direction (`above`, `below`, `centre`, `center`); default `belo`

`XLPOSITION` = *string tokens*	Position of the label in the x-direction (`centre`, `center`, `left`, `right`); default `left`
`PENLABEL` = *scalars*	Pen to use for each label

The `DREFERENCELINE` procedure adds reference lines to a plot. The window containing ther plot specified by the `WINDOW` option. The `ORIENTATION` option controls whether the lines are horizontal (i.e. parallel to the x-axis) or vertical (i.e. parallel to the y-axis).

The `POSITION` parameter defines the position of each line, on the y-axis for a horizontal line, or the x-axis for a vertical line. The `PEN` parameter can specify the pen to use for the line. If this is not set, pen 255 is used as a default, having first been defined to draw continuous light grey lines in 0.75 thickness.

The `LABEL` parameter allows you to plot a label alongside the line. Its position is specified by the `YLPOSITION` and `XLPOSITION` parameters. The pen to use can be specified by the `PENLABEL` parameter. If this is not set, pen 256 is used as a default, having first been defined to omit any symbol and use the colour black.

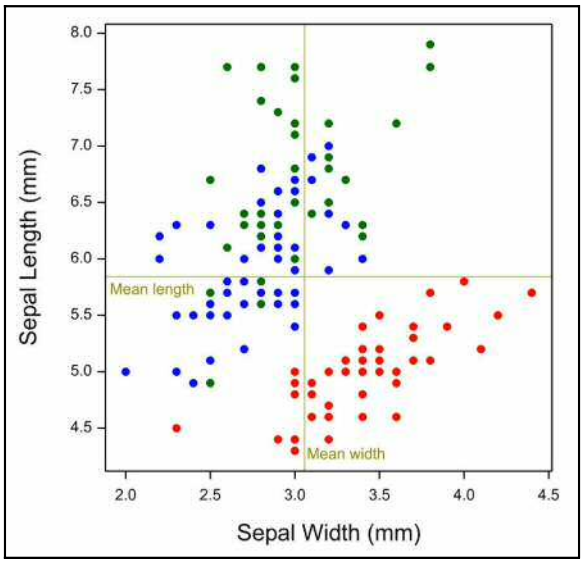

Figure 6.7.2

Example 6.7.2 continues the plotting of Fisher's Iris data from Example 6.2.2. In Figure 6.7.2, it first plots petal length against petal width, and the uses `DREFERENCELINE` to put reference lines through the mean of each variate.

Example 6.7.2

```
PEN            [RESET=yes] 1...3; CSYMBOL='red','blue','darkgreen';\
               CFILL='red','blue','darkgreen'; SYMBOL='circle'
DGRAPH         [KEYWINDOW=0] Y=Slength; X=Swidth; PEN=Species
CALCULATE      mpl,mpw = MEAN(Slength,Swidth)
PEN            4,5; CLINE='olive'; CSYMBOL='olive'
DREFERENCELINE [ORIENTATION=horizontal] mpl; LABEL='Mean length';\
               PEN=4; PENLABEL=5
DREFERENCELINE [ORIENTATION=vertical] mpw; LABEL='Mean width';\
               PEN=4; PENLABEL=5; XLPOSITION=right
```

6.7.3 The `DARROW` procedure

`DARROW` procedure

Adds arrows to an existing plot (D. B. Baird).

Options

`WINDOW` = *scalar*	Window number for the graphs; default 3
`COORDINATETYPE` = *string token*	Type of coordinate to use for the locations of the arrows (`frame`, `graph`); default `grap`
`YUPPER` = *scalar*	Maximum vertical coordinate in the frame; default 1
`XUPPER` = *scalar*	Maximum horizontal coordinate in the frame; default 1

`ISTYLE` = *string token*	The type of symbol at the start of the arrow (`none`, `open`, `closed`, `circle`); default `none`
`ESTYLE` = *string token*	The type of symbol at the end of the arrow (`none`, `open`, `closed`, `circle`); default `open`
`ISIZE` = *scalar*	The size of the symbol at the start of the arrow; default 1
`ESIZE` = *scalar*	The size of the symbol at the end of the arrow; default 1
`IANGLE` = *scalar*	The angle in degrees of the starting arrowhead when `ISTYLE` is `open` or `closed`; default 45
`EANGLE` = *scalar*	The angle in degrees of the ending arrowhead when `ESTYLE` is `open` or `closed`; default 45
`LAYER` = *scalar*	The plot layer for the arrows; default is a new layer above the previous plot items

Parameters

`IY` = *variates, scalars* or *factors*	The starting y-positions of the arrows
`IX` = *variates, scalars* or *factors*	The starting x-positions of the arrows
`EY` = *variates, scalars* or *factors*	The ending y-position of the arrows
`EX` = *variates, scalars* or *factors*	The ending x-position of the arrows
`COLOUR` = *variates, scalars*, *texts* or *factors*	Colour of the arrows; default `'black'`
`LINESTYLE` = *variates, scalars* or *factors*	Linestyle of the line in the arrows; default 1
`THICKNESS` = *variates, scalars* or *factors*	Thickness of the line in the arrows; default 1
`TRANSPARENCY` = *variates, scalars* or *factors*	Transparency of the arrows; default 0

`DARROW` adds arrows or lines to existing plots. The coordinates defining the start and end points of the arrows are specified by the `IY`, `IX`, `EY` and `EX` parameters. The `COLOUR`, `LINESTYLE`, `THICKNESS` and `TRANSPARENCY` parameters specify the colour, linestyle, thickness and transparency (`0` = opaque - `255` = completely transparent) of each arrow. These can supply a single value, if all the arrows are to have the same attribute; otherwise, they should supply a structure of the same length as the `IY` vector.

The `WINDOW` option specifies the number of the window containing the existing plot. By default, the points defining the arrows are specified in terms of the x- and y-axes in the plot. However, you can set option `COORDINATETYPE=frame` to define the points relative to the frame. The maximum size of the frame is then defined by the `XUPPER` and `YUPPER` options.

The `ISTYLE` and `ESTYLE` options control the symbols at the start and end of the arrow, respectively. Similarly, the `ISIZE` and `ESIZE` options define the symbol sizes. The `IANGLE` and `EANGLE` options control the angle between the two sides at the start and end of the arrowheads. Setting the angles to values greater than 180 (e.g. 315) reverses the direction of the arrowheads.

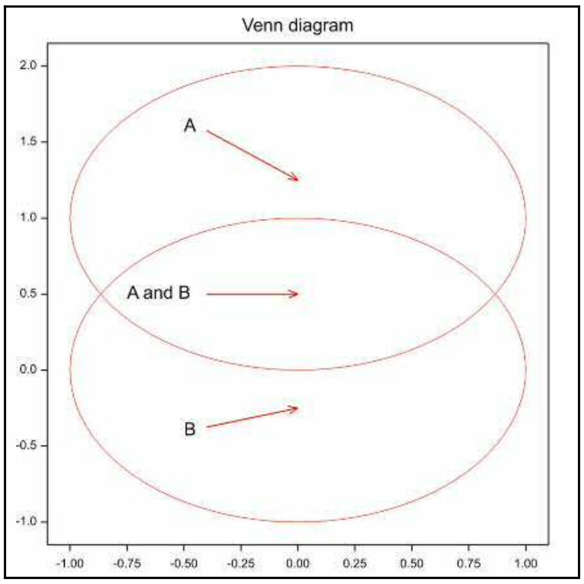

Figure 6.7.3

The `LAYER` option controls which of the existing items in the plot will be overlaid by the arrows. By default, they overlay all the previous items.

Example 6.7.3 replots the Venn diagram in Figure 6.7.1, but now with arrows from the plotted text to the relevant zones; see Figure 6.7.3.

Example 6.7.3

```
  12  DGRAPH     [TITLE='Venn diagram'; WINDOW=3; KEY=0] Y1,Y2; X; PEN=2
  13  DTEXT      [WINDOW=3] Y=1.6,0.5,-0.4; X=-0.5,-0.75,-0.5;\
  14             TEXT='A','A and B','B'; PEN=1
  15  DARROW     IY=1.575,0.5,-0.375; IX=-0.4; EY=1.25,0.5,-0.25; EX=0;\
  16             COLOUR='red'; THICKNESS=1.5
```

6.7.4 The `DERRORBAR` procedure

`DERRORBAR` procedure

Adds error bars to a graph (R.W. Payne).

Options

`ORIENTATION` = *string token*	Direction of the line (`horizontal`, `vertical`); default `vert`
`BARCAPWIDTH` = *scalars*	Width of the cap drawn at the ends of the error bar; default 1
`WINDOW` = *scalar*	Window in which to draw the bar; default 1
`KEYWINDOW` = *scalar*	Window number for the key (zero for no key); default 2

Parameters

`BARLENGTH` = *scalars*	Lengths of the bars
`Y` = *identifiers*	Vertical coordinates for the midpoints of the bars
`X` = *identifiers*	Horizontal coordinates for the midpoints of the bars
`PEN` = *scalars*	Pen to use for each bar
`LABEL` = *texts*	Text to plot alongside each bar
`YLPOSITION` = *string tokens*	Position of each label in the y-direction (`above`, `below`, `centre`, `center`); default `belo`
`XLPOSITION` = *string tokens*	Position of each label in the x-direction (`centre`, `center`, `left`, `right`); default `righ`
`PENLABEL` = *scalars*	Pen to use for each label
`DESCRIPTION` = *texts*	Annotation for the key

The `DERRORBAR` procedure plots error bars on a graph. The window containing the graph is specified by the `WINDOW` option (default 1). The `ORIENTATION` option controls whether the bars are horizontal (i.e. parallel to the x-axis) or vertical (i.e. parallel to the y-axis).

The `BARLENGTH` parameter defines the length of each bar, on the y-axis for a vertical line, or the x-axis for a horizontal line. The positions of their midpoints are specified by the `Y` and `X` parameters. If these are not set, a vertical bar will be plotted just inside the left-hand side of the window, and a horizontal bar will be plotted at the bottom of the window. The `PEN` parameter can specify the pen to use for each bar. If this is not set, pen 255 is used as a default, having first been defined to draw continuous black lines. The `BARCAPWIDTH` option specifies the size of the "caps" drawn at the ends of the bars.

The `LABEL` parameter allows you to plot a label alongside each bar. Its position is specified by the `YLPOSITION` and `XLPOSITION` parameters. The pen to use can be specified by the `PENLABEL` parameter. If this is not set, pen 256 is used as a default, having first been defined to omit any symbol and use the colour black.

The DESCRIPTION parameter can supply annotation to add to the key for each bar. The window for the key is specified by the KEYWINDOW option (default 2).

Example 6.7.4 adds bars, representing the standard deviations, to the graph of Fisher's Iris data in Figure 6.7.2. These can be seen in the final plot of these data, in Figure 6.7.5, where a key has also been added.

Example 6.7.4

```
CALCULATE Wsd,Lsd = SQRT(VARIANCE(Slength,Swidth))
DERRORBAR Wsd; Y=7; LABEL='s.d.'; YLPOSITION=centre
&         [ORIENTATION=horizontal] Lsd; X=4; LABEL='s.d.';\
          YLPOSITION=above; XLPOSITION=centre
```

6.7.5 The DKEY procedure

DKEY procedure

Adds a key to a graph (D.B. Baird & V.M. Cave).

Options

WINDOW = *scalar*	Window in which to draw the key; default 2
NCOLUMNS = *scalar*	Number of columns forming the grid in which the key is displayed; default * (i.e. set automatically)
NROWS = *scalar*	Number of rows forming the grid in which the key is displayed; default * (i.e. set automatically)
TITLE = *text*	Title for the key
PENTITLE = *scalar*	Pen used to write the title of the key; default is that set for the window in which the key is plotted
PENLABELS = *variate*	Pens to use to plot the labels; default is to plot the labels using the settings of LFONT, LSIZE and LCOLOUR
TPOSITION = *string*	Position of the title (inside, outside, left, centre, center, right); default cent, outs
ORDER = *string*	Order in which to fill the key's row by column grid (rows, columns); default rows
LSIZE = *scalar*	Relative size of the labels; default 1
LFONT = *scalar* or *text*	Font to use for the labels; default 1
LCOLOUR = *scalar* or *text*	Colour used to write the labels; default 'black'
XLOFFSET = *scalar* or *variate*	Offset in the x-direction between the items (i.e. symbols/lines) and labels in the key; default 0
COLSPACING = *string*	Column spacing (equal, unequal); default equa
ROWGAP = *scalar*	Multiplier for gaps between rows; default 1
COLGAP = *scalar*	Multiplier for gaps between columns; default 1
BORDER = *string*	Border around the key (fit, given, none); default fit
CBORDER = *string*	Colour for the border around the key; default 'black'

Parameters

DESCRIPTIONS = *texts*	Labels for the key
PEN = *variates*	Pens to use for the items in the key; default uses the integers 1, 2 ...
METHOD = *texts*	Method for plotting the items in the key (fill, point, line, both, none); default is to use the method defined

for the corresponding PEN

SYMBOL = *variates*, *scalars*, *factors* or *texts*
Symbols to be drawn in the key; default is to use those specified by PEN

COLOUR = *variates*, *scalars*, *factors* or *texts*
Colours of lines, or of filled areas when METHOD='fill'; default is to use those specified by PEN

CSYMBOL = *variates*, *scalars*, *factors* or *texts*
Colours of symbols; default is to use those specified by PEN

CFILL = *variates*, *scalars*, *factors* or *texts*
Colours used to fill hollow symbols; default is to use those specified by PEN

SIZEMULTIPLIER = *variates*, *scalars* or *factors*
Relative sizes of symbols and filled area; default is to use those specified by PEN

LINESTYLE = *variates*, *scalars* or *factors*
Numbers or names of the linestyles to use; default is to use those specified by PEN

THICKNESS = *variates*, *scalars* or *factors*
Thicknesses of the lines; default is to use those specified by PEN

TRANSPARENCY = *variates*, *scalars* or *factors*
Transparencies of the filled areas when METHOD='fill'; default is to use those specified by PEN

The DKEY procedure provides a more flexible way of providing a key for a plot, than the standard facilities provided by the ordinary plotting commands. The standard keys can be suppressed by setting the option KEYWINDOW in those commands to zero.

The labels to appear in the key must be supplied as a text structure by the DESCRIPTIONS parameter. The number of labels defines the number of items *n* to appear in the key. The appearance of the labels (size, font and colour) can be controlled either by the PENLABELS option by or the LSIZE, LFONT and LCOLOUR options. PENLABELS can supply a variate, with *n* values, to define the pens to use for the labels.

If PENLABELS is not set, the labels are all written in the same style, using the settings of the LSIZE, LFONT and LCOLOUR options. The LSIZE option modifies the size of the labels, by specifying a value by which the default size is to be multiplied; default 1. The LFONT option specifies the font to use for the labels. This can be set either to a text containing the name of a font family, or to a scalar containing an integer between 1 and 25. The default is to use the *default graphics font* (6.9.12). The LCOLOUR option specifies the colour for the labels (6.9.9). The default is 'black'.

The METHOD parameter supplies a text defining the types of item to be plotted in the key. The text can contain a single string if all the items are to be displayed in the same way, or a string for each item if they are to be displayed differently. The possible strings are

'point'	for points,
'line'	for lines,
'both'	for points and lines,
'fill'	for filled rectangles, and
'none'	to prevent an item from being plotted.

The default is to use the method defined for the corresponding PEN.

The appearance of the items (symbol type, colour, size, linestyle, line thickness and transparency) can be controlled by specifying the pens to be used to plot them by the PEN parameter. The default is to use pens 1 ... *n*.

Alternatively, you can set the appearance of the items explicitly, by using the parameters SYMBOL, COLOUR, CSYMBOL, CFILL, SIZEMULTIPLIER, LINESTYLE, THICKNESS and TRANSPARENCY. (These override the settings from PEN.) For each of these parameters, you can supply either a single value or a structure with *n* values (one for each item).

The SYMBOL parameter defines the symbols for items that are displayed as points, or as both points and lines (6.9.8).

The COLOUR, CSYMBOL and CFILL parameters specify the colours to be used for the items. The COLOUR parameter defines the colours of lines and filled areas. The CSYMBOL parameter defines the colours used for symbols. The CFILL parameter defines the colours used for filling areas inside hollow symbols (6.9.9). The transparency of a filled area can be set using the TRANSPARENCY parameter. This can be set either to a scalar or variate containing values between 0 (opaque) and 255 (completely transparent), or to factor with at most 255 levels.

The SIZEMULTIPLIER parameter can modify the size of symbols and filled areas, by specifying a value by which the default size is to be multiplied. Either a scalar, variate or factor can be supplied. The LINESTYLE parameter defines what sort of line is drawn, for example, a solid, dotted or dashed line (6.9.8). The THICKNESS parameter can modify the thickness of lines, by specifying a value by which the standard thickness is to be multiplied. Either a scalar, variate or factor may be supplied.

The WINDOW, NCOLUMNS, NROWS, ORDER, XLOFFSET, COLSPACING, ROWGAP, COLGAP, BORDER and CBORDER options control the layout of the key. The WINDOW option specifies the window in which the key is drawn; default 2. The number of rows and columns, forming the grid in which the key is arranged, can be set by the NROWS and NCOLUMNS options, respectively. If these are not set, an appropriate grid is constructed automatically. The order in which the items fill the grid is determined by the ORDER option. The default, ORDER=rows, fills the grid row by row. Alternatively ORDER=columns fills the grid column by column. The COLSPACING option specifies whether or not the columns of the grid are equally spaced (equal and unequal, respectively); default equal. The ROWGAP and COLGAP options control the sizes of the gaps between rows and columns, respectively. The distance between the items and labels can be adjusted by the XLOFFSET option. Each label in the grid can be individually offset by suppling a variate with *n* values. When a single value is supplied, a common offset is applied to all labels in the grid. The BORDER option controls the border drawn around the key. The default, BORDER=fit, draws a border fitted to the key. When BORDER=given, the border frames the window (and the key is drawn so that it occupies the entire window). Finally, if BORDER=none, no border is drawn. The CBORDER option specifies the colour for the border, when one is drawn around the key; default 'black'.

The TITLE option can provide a title for the key. The pen for the title can be set by the PENTITLE option. The default is to use the pen defined for the window in which the key is plotted. The TPOSITION parameter specifies the position of the title: either inside or outside the border with left, right or centre justification. The default is to centre the title outside the border of the key.

DKEY takes account of restrictions on DESCRIPTIONS, PEN, PENLABELS and XLOFFSET. However, the parameters METHOD, SYMBOL, COLOUR, CSYMBOL, CFILL, SIZEMULTIPLIER, LINESTYLE, THICKNESS and TRANSPARENCY must not be restricted.

Example 6.7.5adds a key to the scatter plot in Figure 6.7.2. The `KEYWINDOW` option of `DGRAPH` was set to zero, to suppress the standard key, when the graph was plotted in Example 6.7.2. The `FRAME` command in line 175 defines window 6, alongside the scatter plot to use for the key. `DKEY` (lines 176-178) plots a key in a single column, with no border and no lines or symbols. An offset of `-6` is specified, to move the labels across to use the space that would have been used for the symbols and lines. Notice that we have used the typesetting command `~i{}` to put the labels into italic font; see 1.4.2.

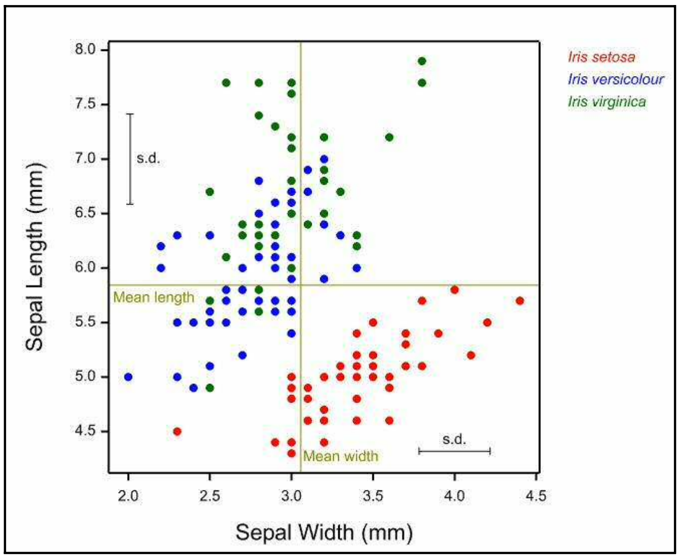

Figure 6.7.5

Example 6.7.5

```
175   FRAME  6; YLOWER=0; YUPPER=1; XLOWER=0.72; XUPPER=1
176   DKEY   [WINDOW=6; NCOLUMNS=1; PENLABELS=!(1...3); BORDER=none; XLOFFSET=-6]\
177          !T('~i{Iris setosa}','~i{Iris versicolour}','~i{Iris virginica}');\
178          METHOD='none'
```

6.8 Multiple high-resolution plots

Many Genstat graphics commands have a `SCREEN` option, which can be set to `keep` to enable you to add new information to the current display. The output from each command is drawn in one or more graphics *windows*. There are 256 windows (see `FRAME`, 6.9.3). They are independent of one another, and most graphics devices allow you to display them simultaneously on the same graphics screen. On most devices you can also have windows that overlap or contain others. So you can plot to a sequence of windows (keeping the current display), and build up a multiple display with different graphs in adjacent windows. Several Genstat procedures use this facility: for example trellis plots are described in 6.8.3, and scatter-plot matrices in 6.8.4.

Alternatively, you may be able to plot new information in an existing window, and build up a complicated picture in several stages. However, there are limitations on what a single window can contain: you can use `DGRAPH` (6.2.1) any number of times, but you can use no more than one other command, which may be either `DHISTOGRAM` (6.3.1), `DCONTOUR` (6.4.1) or `DSHADE` (6.4.2). This approach is used in graphics procedures, like `BOXPLOT` (2:2.2.2) or `DDENDROGRAM` (2:6.17.5), and is illustrated in Example 6.8 and Figure 6.8 where a graph and a histogram are plotted in window 3.

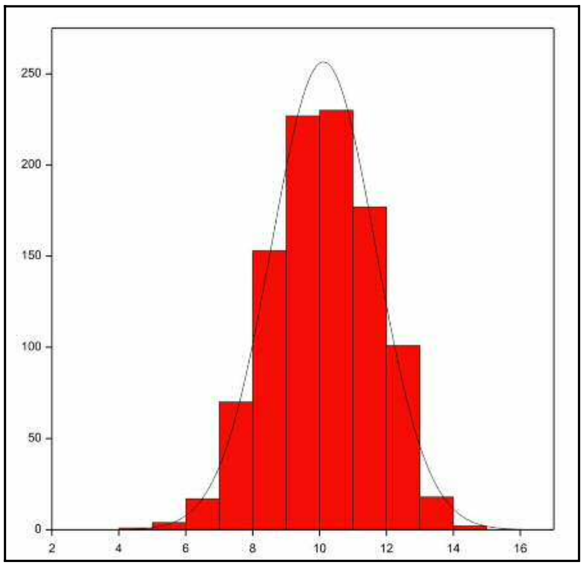

Figure 6.8

Example 6.8

```
   2  READ Data

    Identifier   Minimum      Mean   Maximum    Values   Missing
          Data     4.390     10.11     14.55      1000         0

 104  VARIATE    [VALUES=2.0,2.1...17.0] X
 105  CALCULATE  Mu = MEAN(Data)
 106  &          Sigma = SQRT(VARIANCE(Data))
 107  &          Y = 1 / (SQRT(2*C('PI'))*Sigma) * EXP(-0.5*((X-Mu)/Sigma)**2)
 108  &          Y = Y * NVALUES(Data)
 109  PEN        1; METHOD=monotonic; SYMBOL=0
 110  XAXIS      3; LOWER=2.0; UPPER=17.0
 111  YAXIS      3; LOWER=0.0; UPPER=275
 112  DGRAPH     [WINDOW=3; KEYWINDOW=0] Y; X
 113  VARIATE    [VALUES=3...16] Limits
 114  DHISTOGRAM [WINDOW=3; KEYWINDOW=0; LIMITS=Limits; SCREEN=keep] Data
```

6.8.1 Clearing the graphics screen: the DCLEAR directive

DCLEAR directive

Clears a graphics screen.

Options

`DEVICE` = *scalar*	Device whose screen is to be cleared; default is to clear the screen of the current graphics device
`ENDACTION` = *string token*	Action to be taken after clearing the screen (`continue`, `pause`); default `*` uses the setting from the last `DEVICE` statement

No parameters

When generating displays using a sequence of graphics commands, it may be convenient to clear the screen at the outset. Then the subsequent commands can all have option `SCREEN=keep`, which will simplify the programming particularly if they are in a FOR loop (5.2.1). Thus DCLEAR allows you to clear the screen of a graphics device so that the next plot produced on this device by any of the high-resolution commands will be drawn onto an empty screen. All information about the current display, for example axis mappings, is also cleared from memory. The DEVICE option indicates the device to be cleared; by default this is the current graphics device (as set by the DEVICE directive). The ENDACTION option controls what happens after clearing the screen. The default action is the setting specified by the most recent DEVICE statement.

6.8.2 Sequences of high-resolution plots

DSTART directive

Starts a sequence of related high-resolution plots.

Options

TITLE = *text*	Overall title for the plots
PEN = *scalar*	Pen to use for the title; if this is not set, pen −12 is used

DFINISH directive

Ends a sequence of related high-resolution plots.

No options or parameters

The most efficient way of generating a composite display is to define an explicit sequence of plots. The start of the sequence is indicated by a DSTART command. The TITLE option can specify an overall title, and PEN can specify the pen to use. If PEN is not set, the title is plotted using pen −12.

During the sequence the information from each graphics command is accumulated until Genstat finds a DFINISH command. Genstat then clears the screen and generates the display. This improves efficiency, as no plotting takes place until the display is complete. It also simplifies programming as the SCREEN option is irrelevant; any settings of the SCREEN option in the plotting directives during the sequence are ignored.

6.8.3 Trellis plots: the TRELLIS procedure

TRELLIS procedure

Does a trellis plot (S.J. Welham & S.A. Harding).

Options

GROUPS = *factors* or *variate*	Factors or variate defining the classification for the plots
GMETHOD = *string token*	Determines the method used to partition the range when GROUPS is set to a variate (equalspacing, quantiles, distinct, limits); default equal
NGROUPS = *scalar*	Determines the number of plots to be formed when GROUPS is set to a variate and GMETHOD is set to quantiles or equalspacing
LIMITS = *variate*	Limits to use to form groups from a GROUPS variate when GMETHOD=limits
OVERLAP = *scalar*	Proportion by which a GROUPS variate should overlap between plots (scalar in range 0 - 0.5); default 0
OMITEMPTY = *string token*	Whether to omit all empty plots from the array (all), or

	omit levels of a GROUPS factor where all plots are empty (levels), or keep all plots in the array (none); default level
PENGROUP = *factors*	Defines factor combinations to be plotted in different colours, note that the number of colours available may differ between devices
NROWS = *scalar*	Specifies number of rows of plots to appear on one page; default determined automatically from GROUPS
NCOLUMNS = *scalar*	Specifies number of columns of plots to appear on one page; default determined automatically from GROUPS
TITLE = *text*	Supplies a title for the plot
FIRSTPICTURE = *string token*	Whether to put the first picture at bottom or top left of the grid (bottomleft, topleft); default topl
TMETHOD = *string token*	Whether to give plot titles as factor names with labels or just labels (names, labels); default names
YTITLE = *text*	Supplies an overall y-axis title
XTITLE = *text*	Supplies an overall x-axis title
YMARGIN = *scalar*	Relative size of margins for the y-axis labels on individual plots; default 0.04
XMARGIN = *scalar*	Relative size of margins for the x-axis labels on individual plots; default 0.04
TMARGIN = *scalar*	Relative size of margin for titles of individual plots; default 0.04
PENSIZE = *scalar*	Proportionate adjustment to the pen size for individual plot titles and axis labels; default 1
USEPENS = *string token*	Whether to use current pen definitions in the procedure (no, yes); default no
USEAXES = *string token*	Which aspects of the current axis definitions of window 1 to use (none, limits, style, marks, mpositions, nsubticks, transform); default none
NRMAX = *scalar*	Maximum number of rows on page; default 8 for a square frame, 7 for a landscape frame and 10 for a portrait frame
NCMAX = *scalar*	Maximum number of columns on page; default 8 for a square frame, 10 for a landscape frame and 7 for a portrait frame
KEYHEIGHT = *scalar*	Space in y-direction to use for key (0 to suppress key); default * i.e. determined automatically
YPENMETHOD = *string token*	Whether to use the same or different pens for each y-variate (different, same); default diff
FRAMESHAPE = *string token*	Shape of the plotting frame (landscape, portrait, square); default squa

Parameters

Y = *variates*	Y-values of the data to be plotted
X = *variates* or *factors*	X-values of the data to be plotted
METHOD = *string tokens*	Type of plot (point, line, mean, median, histogram, boxplot, spline, schematicboxplot); default poin
DESCRIPTION = *texts*	Annotation for key

TRELLIS plots one or more y-variates for each level generated by the GROUPS option, and arranges these plots in a grid (or trellis) arrangement on the page.

The data to be plotted are specified using the Y parameter. If more than one variate is specified, these will all be displayed on the same plots. This means that e.g. data points can be plotted with means. The type and method of plotting (points, lines, mean values, medians, histograms, boxplots or splines) is specified using the METHOD parameter. The default is METHOD=point. For methods point, line, mean, median and spline, a graph is produced of y-variates against x-variates, which are specified using the X parameter. When METHOD is set to mean or median, a line is drawn to join the mean or median data values at each value of the x-variate for each level of PENGROUP. In any of these cases, if PENGROUPS is set to one or more factors, a different pen will be used for each of the levels of the combined factors. By default, the pen numbers are incremented so that a different set of pens is used for each y-variate. Alternatively, you can set option YPENMETHOD=same, to use the same set for each one.

When METHOD=histogram, a histogram of the data values is drawn in each plot. In this case, options NGROUPS and LIMITS can be used to specify the number of groups in the histogram or the group limits, respectively. If more than one y-variate is specified, parallel histograms will be drawn for the variates. The PENGROUPS option is ignored when METHOD=histogram.

When METHOD=boxplot, a boxplot of the data values is drawn in each plot. Alternatively, you can set METHOD=schematicboxplot to obtain a schematic boxplot, which displays individual outlying points as well as the box (see 2:2.2.2). If you set the PENGROUP option, parallel box plots (one for each level of PENGROUPS) are drawn within every plot. You can also obtain parallel box plots by supplying several y-variates, which are then plotted in parallel in every plot. However, you cannot simultaneously specify several y-variates and set the PENGROUPS option.

The division of the data into separate plots is determined by the setting of the GROUPS option. This can be set to one or more factors, indicating that a separate plot should be drawn for each combinations of the factor levels.

The OMITEMPTY option controls what happens if there are no data for some combinations. The default setting levels omits complete levels of any factor for which there are no data points, while the setting all omits all empty plots, i.e. plots where there are no data points. OMITEMPTY=none displays all plots regardless of whether or not they contain any data points.

If the GROUPS option is set to a variate, the plots will show the values of the data for different intervals in the range of the GROUPS variate. The GMETHOD, NGROUPS, LIMITS and OVERLAP options determine how many plots are displayed, and which data points they contain. The default option of GMETHOD is equalspacing. The distinct setting of GMETHOD converts the variate into a factor with a level (and thus a plot) for each distinct value of the variate. With equalspacing, the groups are defined by dividing the range of the GROUPS variate into the required number of intervals of equal length; while with quantiles, the intervals are defined so that each has an equal number of points, according to the ordering of the GROUPS variate. When GMETHOD is set to equalspacing or to quantiles, the number of groups to form can be specified by the NGROUPS option; if NGROUPS is not set, TRELLIS sets the number to the square root of the number of data values, or to the number of distinct values if this is smaller. Finally, when GMETHOD=limits, the LIMITS option specifies boundaries between the intervals; the first group then contains all data points with values of the GROUPS variate less than the first limit, the second group has all values greater than or equal to the first limit but less than the second limit, and so on.

The OVERLAP option allows the intervals of the GROUPS to overlap. The default overlap is 0, so there is no overlap between plots. If OVERLAP is set to 0.1, then 10% of the points (for PARTITION=quantiles) or 10% of the range (for PARTITION=equalspacing) will be in common between neighbouring plots. OVERLAP can be set anywhere in the range 0 (for no overlap) to 0.5.

The `FRAMESHAPE` option specifies the shape of the graphics frame, with settings:

`landscape`	for a frame of size 1.4 × 1.0 i.e. wider in the x- than the y-direction,
`portrait`	for a frame of size 1.0 × 1.4 i.e. wider in the y- than the x-direction,
`square`	for a frame of size 1.0 × 1.0.

Some graphics devices do not support the use of device coordinates greater than 1.0, so the default is `FRAMESHAPE=square`. (See 6.9.1 and 6.9.3 for more information.)

The default layout on the page can be changed by using `NROWS` and `NCOLUMNS` to specify the number of rows of plots on the page, and the number of columns of plots across the page, respectively. By default the layout is arranged so that the area of the page used for plotting is maximized, with a maximum of 8 rows and 8 columns of plots for a square frame, 7 rows and 10 columns for a landscape frame, and 10 rows and 7 columns for a portrait frame. Options `NRMAX` and `NCMAX` can be used to override these default maximum numbers of rows and columns of plots, so that more can be produced on a page.

An overall title can be put at the head of each page using the `TITLE` option, and overall titles for the y- and x- axes can be specified using the `YTITLE` and `XTITLE` options respectively. By default the plots start at the top left of the page, but you can set option `FIRSTPICTURE=bottomleft` to start at the bottom left. When `GROUPS` is set to one or more factors, the plot titles are constructed by default with the factor name and label/level, but this can be restricted to just the label/level by setting option `TMETHOD=label`.

The margins and pen size are set to give a reasonable picture on the Windows PC implementation, but can be adjusted using options `YMARGIN` (space for y-axis labels), `XMARGIN` (space for x-axis labels), `TMARGIN` (space for plot titles) and `PENSIZE` (pen size for axis markings and plot titles).

By default the pen and axes attributes are determined automatically within the procedure. Some predefined attributes can be used, as indicated by the `USEPENS` and `USEAXES` options. Setting `USEPENS` to `yes`, requests all current pen definitions (for pens 1-29) to be used.

You can specify various aspects of the axes, by defining them for window 1, and indicating that they are to be used by setting the `USEAXES` option. The following settings are available:

`limits`	y- and x-axis limits (`LOWER` and `UPPER` parameters of `XAXIS` and `YAXIS`);
`style`	axis styles (`ACTION` parameter of `XAXIS` and `YAXIS`, together with the `GRID` option and `BOX` parameter of `FRAME`);
`marks`	location and labelling of the tick marks (`MARKS`, `LABELS`, `LDIRECTION`, `LROTATION`, `DECIMALS`, `DREPRESENTATION`, and `VREPRESENTATION` parameters of `XAXIS` and `YAXIS`);
`mpositions`	positions of the tick marks (`MPOSITION` parameter of `XAXIS` and `YAXIS`); and
`nsubticks`	number of subticks per interval (`NSUBTICKS` parameter of `XAXIS` and `YAXIS`); and
`transform`	axis transformations (`TRANSFORM` parameter of `XAXIS` and `YAXIS`).

`TRELLIS` includes a key on each graphics page for plots other than boxplots if each window of the trellis contains more than plot (i.e. if there is more than one `Y` variate, or there is a `PENGROUPS` factor with more than one level). You can use the `KEYHEIGHT` option to control the size of the key in the y-direction, and setting this to zero will suppress the key. The `DESCRIPTION` parameter can be used to supply annotation for the key, in the same way as in the `DGRAPH` directive.

Example 6.8.3 uses `TRELLIS` to study the relationship between amounts of sulphur in the air and weather variables. Sulphur is plotted against wind speed for every combination of factors `Winddirection` and `Rain`.

Example 6.8.3

```
   2  " Comparison of air pollution and weather variables:
  -3    sulphur levels against wind speed, wind direction, and rain."
   4  FACTOR [LABELS=!t(N,NE,E,SE,S,SW,W,NW)] Direction
   5  &      [LABELS=!t(no,yes)] Rain
   6  READ   Sulphur,Speed,Direction,Rain; FREPRESENTATION=labels

    Identifier   Minimum      Mean   Maximum    Values   Missing
       Sulphur    0.0000     10.46     49.00       114         0    Skew
         Speed    0.5000     10.31     22.70       114         1

    Identifier    Values   Missing    Levels
     Direction       114         1         8
          Rain       114         0         2

  36  TRELLIS [GROUPS=Direction,Rain; NROW=4;\
  37          TITLE='Sulphur versus wind speed';\
  38          YTITLE='Sulphur measurements';\
  39          XTITLE='Wind speed (km/h)'] Sulphur; Speed
```

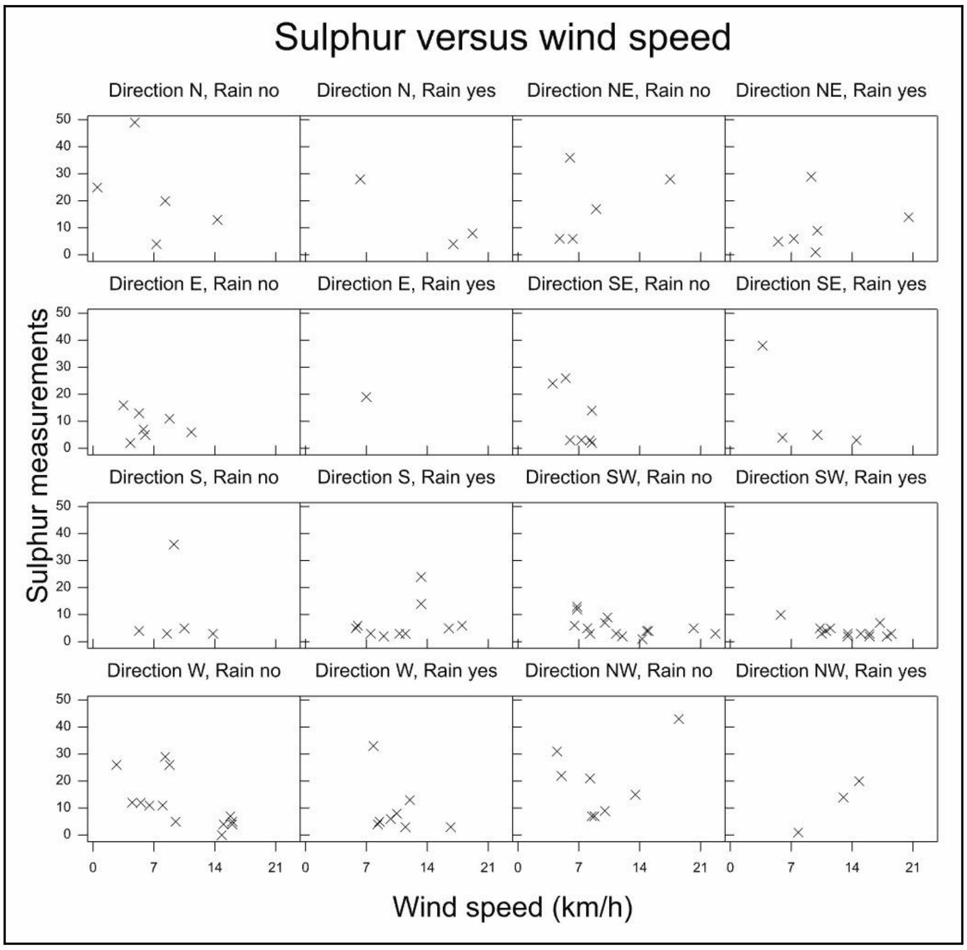

Figure 6.8.3

6.8.4 Scatter-plot matrices: the DMSCATTER procedure

DMSCATTER procedure

Produces a scatter-plot matrix for one or two sets of variables (J. Ollerton & R.W. Payne).

Options

PLOT = *string tokens*	Additional information to include in the scatter plots (correlation, histograms, boxplots, densities, dothistograms); default *
SCALING = *string token*	How to scale the x- and y-axes (common, equal, none); default none
PEN = *scalar* or *variate* or *factor*	Pens to plot the scatter plots; default 1
PENHISTOGRAM = *scalar*	Pens to plot the histograms; if PEN is a factor the default plots the histogram for each group separately using the pen used for that group in the scatter plots, otherwise the default is to use pen 2
PENCORRELATION = *scalar*	Pen to use to write the correlations; default 1
PENTITLE = *scalar*	Pen to use to write the axis titles; default uses the pens currently defined for the axes in the windows that are used for the plots
PENAXIS = *scalar*	Pen to use to draw the axes; default uses the currently defined pens
PENLABELS = *scalar*	Pen to use to write the axis labels; default uses the currently defined pens
NROWS = *scalar*	Number of rows of graphs to put in a single frame (i.e. page); default puts them all in one frame
NCOLUMNS = *scalar*	Number of columns of graphs to put in a single frame; default uses the same value as NROWS
ASPECTRATIO = *scalar*	Ratio of the length of the y-axis to the length of the x-axis in each graph
FRAMESHAPE = *string token*	Shape of the plotting frame (landscape, portrait, square); default squa
MARGINSIZE = *scalar*	Specifies the size of the margins at the bottom and left-hand edge of the frame

Parameters

Y = *pointers*	Each pointer contains a set of variates and/or factors to be plotted
X = *pointers*	Each pointer contains a set of variates and/or factors to be plotted as the x-variables in a rectangular scatter-plot matrix; if unset Y specifies both the x-variables and y-variables for a symmetric scatter-plot matrix
TITLE = *texts*	Overall title for the plot
YTITLES = *texts*	Labels for the axes for the Y variates and factors, to use instead of their identifiers
XTITLES = *texts*	Labels for the axes for the X variates and factors, to use instead of their identifiers
YMARKS = *variates*, *scalars* or *pointers*	Marks to use on the axes for the Y variates and factors, if any of these contains missing values, the marks and their

	labels are suppressed for that variate or factor
XMARKS = *variates*, *scalars* or *pointers*	
	Marks to use on the axes for the X variates and factors, if any of these contains missing values, the marks and their labels are suppressed for that variate or factor

Procedure DMSCATTER produces two types of scatter-plot matrix, using high-resolution graphics. For a symmetric scatter-plot matrix, the variates and/or factors to be plotted against each other must be specified, in a pointer, by the Y parameter. The scatter-plot contains a lower-triangular array of graphs, one for each pair of variables. Alternatively, for a rectangular scatter-plot matrix, there are two set of the variates and/or factors. The set that defines the y-values for the graphs are specified (in a pointer as before) by the Y parameter, and those that define the x-values for the graphs are specified (also in a pointer) by the X parameter. The scatter-plot now contains a rectangular array of graphs, one for each pair of x- and y-variables. If any of the variates or factors is restricted, only the units not excluded by the restriction will be plotted.

By default the identifiers of the relevant x- and y-variables are used for the titles of the axes at the lower and left-hand edges of the graphics frame (i.e. page). Alternatively, you define your own titles for the y-variables by setting the YTITLES to a text with a value for each Y variate or factor. Similarly, you can use the XTITLES parameter to supply your own titles for the X variates or factors. You can also use the TITLE parameter to supply an overall title.

The YMARKS parameter allows you to specify your own marks for the axes corresponding to the y-variables. (These are then used as the settings of the MARKS parameter of the YAXIS and XAXIS directives.) You can set YMARKS to single variate or scalar, if you want to use the same marks for every y-variable. Alternatively, you can set it to a pointer with a variate or factor for each Y variate or factor, if you want to specify different marks. If any of the variates or scalars contains missing values, the marks and their labels are suppressed on the corresponding axes. You can use the XMARKS parameter similarly, to specify axis marks for the x-variables.

The PEN option specifies the pens to be used to plot the graphs. The setting can be a scalar to plot all the points with the same pen, or a variate or a factor to use different pens. If PEN is set to a factor, a key is included in the plot to identify the correspondence between the pens and the groups. The default is to use pen 1.

The PLOT option allows you to specify extra information to be included in the plot, with settings:

correlation	prints the correlation of the pair of variables in each plot, at the top of the plot;
histograms	plots histograms of the variables down the diagonal of a symmetric scatter-plot matrix, or along the top and down the right-hand side of a rectangular scatter-plot matrix;
boxplots	displays boxplots of the variables down the diagonal of a symmetric scatter-plot matrix, or along the top and down the right-hand side of a rectangular scatter-plot matrix;
densities	displays one-dimensional density plots (or violin plots) of the variables down the diagonal of a symmetric scatter-plot matrix, or along the top and down the right-hand side of a rectangular scatter-plot matrix; and
dothistograms	plots dot histograms of the variables down the diagonal of a symmetric scatter-plot matrix, or along the top and down the right-hand side of a rectangular scatter-plot matrix.

Note, only one of the settings histograms, boxplots, densities, dothistograms is allowed; if more than one is set, the first item the list above is used.

The PENHISTOGRAM option specifies the pens to plot the histograms. If PEN is a set to a factor, the default for `PENHISTOGRAM` plots histogram for each group, using the pen used for that group in the scatter plots. Otherwise the default is to use pen 2. The `PENCORRELATION` option specifies the pen to use to print the correlations; default 1.

The `PENTITLE`, `PENAXIS` and `PENLABELS` options define the pens to use for the titles of the x- and y-axes, for the axes themselves, and for their labels. If any of these is unset, the default is to use the pens already defined for that aspect of the axes in the windows used in the plot.

The `SCALING` option controls the scaling of the x- and y-axes, the settings:

`equal`	uses equal scaling for the x- and y-axes in each graph,
`common`	used exactly the same axes (upper and lower limits as well as scaling) for the axes in all the graphs,
`none`	defines all the axes independently (the default).

By default the plots are square, but you can request rectangular plots by setting the `ASPECTRATIO` option to the required value for the length of the y-axis divided by the length of the x-axis.

The `MARGINSIZE` option specifies the size of the margins at the bottom and left-hand edge of the graphics frame. If this is unset, the margins are defined automatically, using a smaller value if all the axis marks and labels on an edge have been suppressed.

The `FRAMESHAPE` option specifies the shape of the graphics frame, with settings:

`landscape`	for a frame of size 1.4 × 1.0 i.e. wider in the x- than the y-direction,
`portrait`	for a frame of size 1.0 × 1.4 i.e. wider in the y- than the x-direction,
`square`	for a frame of size 1.0 × 1.0.

Some graphics devices do not support the use of device coordinates greater than 1.0, so the default is `FRAMESHAPE=square`. (See 6.9.1 and 6.9.3 for more information.)

By default the graphs are all plotted in a single frame (i.e. page), but you can specify the `NROWS` and `NCOLUMNS` options to split them across several frames. `NROWS` specifies the number of rows of plots to put in a single frame. The default is to fit them all into one frame. `NCOLUMNS` specifies the number of columns of plots to put in one frame. The default is to use the same value as `NROWS`.

An example is shown in Figure 6.8.4, which was generated by Example 6.8.4.

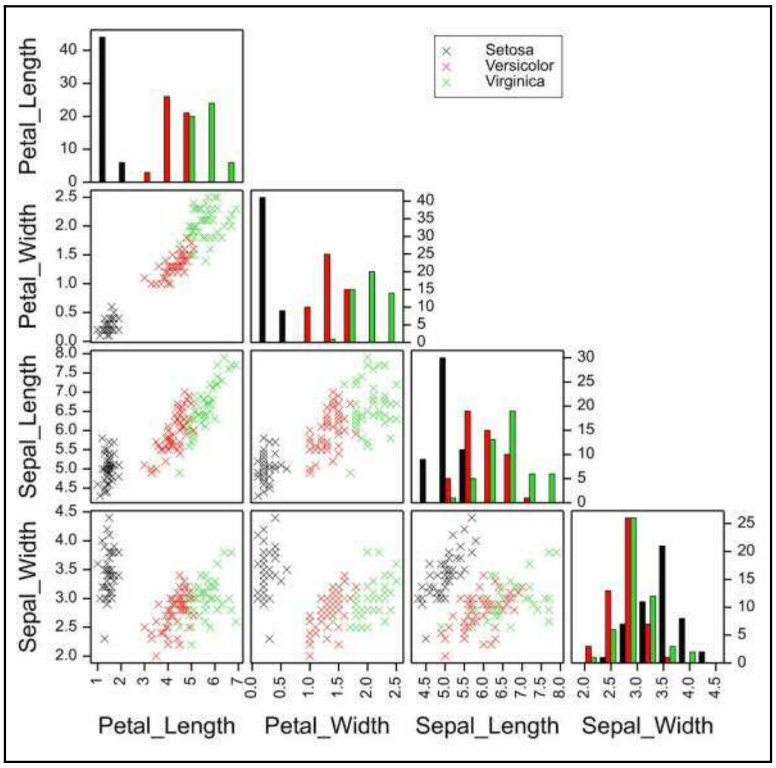

Figure 6.8.4

Example 6.8.4

```
SPLOAD    [PRINT=*] '%GENDIR%/Data/Iris.gsh'
CALCULATE PLength = (Petal_Length-MEAN(Petal_Length))/SQRT(VAR(Petal_Length))
&         PWidth=(Petal_Width-MEAN(Petal_Width))/SQRT(VAR(Petal_Width))
&         SLength = (Sepal_Length-MEAN(Sepal_Length))/SQRT(VAR(Sepal_Length))
&         SWidth = (Sepal_Width-MEAN(Sepal_Width))/SQRT(VAR(Sepal_Width))
POINTER   [VALUES=Petal_Length,Petal_Width,Sepal_Length,Sepal_Width]\
          Measurements
DMSCATTER [PLOT=histograms; PEN=Species] Measurements
```

6.9 The environment for high-resolution graphics

The directives described in the earlier sections of this chapter can display data in various ways. Implicit in all the discussion is the idea of a *graphics environment*, in which the displays are generated. This consists of a choice of graphics devices and a large number of parameters which control the appearance of the output. When you start Genstat an initial environment is created which contains default settings that are designed to be appropriate for the more common types of plot. This section describes the directives that allow you to modify the graphical environment in order to obtain more control over the appearance of your output. The descriptions of the commands earlier in this chapter indicate how the output will appear by default, and how it is affected by changes to the environment. The examples were chosen to illustrate the default display and some of the ways in which it can be modified by directives such as `PEN` (6.9.8), `XAXIS` (6.9.4) and `YAXIS` (6.9.5).

When you produce a high-resolution plot, the pictures are drawn on a *graphical device*, in a *graphical window*, using a *graphical pen*. Output can be produced on only one type of device at any time; however you can switch between different devices during a Genstat session so that, for example, you can experiment with various displays on the screen before sending some output to a file for printing as hard-copy. The device is selected using the DEVICE directive (6.9.1). Note, though, in Genstat *for Windows* you can save the display in these formats directly from the graphics viewer, so the DEVICE statement is needed only if you want to run Genstat as a batch process.

A graphics window is an area of the screen (or page on a plotter) that is used for plotting output. Many such windows can be used within a sequence of statements, so that several graphs may be plotted on a single screen. The position and size of the windows is defined using the FRAME directive (6.9.3). Associated with each window are the attributes of its axes. These control how axes are drawn by directives such as DGRAPH, DHISTOGRAM and DCONTOUR. The XAXIS, YAXIS and ZAXIS directives (6.9.4, 6.9.5 and 6.9.6) can be used to control the various aspects of the axes associated with any specific window. These replace the AXES directive that was used, in releases before 4.2, to set attributes of the x- and y-axes. AXES is retained for compatibility, but it is less powerful than XAXIS and YAXIS. You can also include additional axes in a plot, and these can have oblique directions. They are defined using the AXIS directive (6.9.7), and added into a particular graphics window using the FRAME directive (6.9.3).

Each part of the display is drawn using pens, each of which has attributes such as colour, line style and symbol type. In addition, the pen may be used to control how data is plotted, for example by requesting a straight line or a curve. The PEN directive (6.9.8) is used to set attributes of the different pens to be used in each graph.

The directives that define the environment change only the parameters that are mentioned explicitly; unspecified parameters retain their previous values (which may be the initial defaults). When you start a new job (5.1), the environment is reset to the initial default values. On the other hand, when you use RESUME (3.6.2) to re-start an earlier session, the graphics environment will be loaded from the resume file. However, this does not affect the choice of output device (and associated file) which is preserved in both situations.

As the effects of these directives are additive, you need to keep aware of the current settings, and avoid unwanted side-effects which may occur, for example, if you use a pen that has earlier been modified in a way that is incompatible with its current use. This should not cause problems under ordinary circumstances. However, if you are using graphics in a general program or procedure there are various things you can do to make the graphics self-contained, and avoid side-effects. Each directive that modifies the environment includes a SAVE parameter that enables you to save the current settings of its particular aspect of the environment (frame, axes or pen) after making any modifications specified in the current statement. This enables you to check the current settings and reset particular attributes to their original values after a plot has been produced. The DKEEP directive can be used to obtain additional general information about the graphics devices and environment. The GET and SET directives (5.6.1 and 5.6.2) allow the entire graphics environment to be stored in a pointer and later restored to its original state. For example, in a graphics procedure you might have the following statements:

```
GET    [SPECIAL=Special]
FRAME  1; YLOWER=0.3; YUPPER=0.6; XLOWER=0.3; XUPPER=0.6
YAXIS  1; LOWER=0; UPPER=100; TITLE='Percentages'
PEN    1...4; METHOD=line; LINESTYLE=1...4; SYMBOL=0;\
       COLOUR=1,2,1,2
DGRAPH Percent[1...4]; X; PEN=1...4
SET    [DSAVE=Special['dsave']]
ENDPROCEDURE
```

This can also be done automatically using the RESTORE option of PROCEDURE (5.3.2). Alternatively, you can save the current graphics environment settings to an external file using the DSAVE directive (6.9.11), and reload then later using the DLOAD directive (6.9.11).

Information about the graphics environment can be displayed using the DHELP procedure.

DHELP procedure

Provides information about Genstat graphics (S.A. Harding).

No options

Parameter

TOPIC = *string tokens*	Lists the required graphics topics (current, possible); default poss

6.9.1 The DEVICE directive

DEVICE directive

Switches between (high-resolution) graphics devices.

No options

Parameters

NUMBER = *scalar*	Device number
ENDACTION = *string token*	Action to be taken after completing each plot (continue, pause)
ORIENTATION = *string token*	Orientation of the pictures, if relevant (landscape, portrait); default * retains the current setting for this device
PALETTE = *string token*	How to represent colour (monotone, greyscale, grayscale, colour); default * retains the current setting for this device
RESOLUTION = *scalar*	Specifies the height of the image for hard-copy output, in pixels
ACTION = *string token*	How to create graphs for file types such as .emf, .jpg, .tif or .png (asynchronous, synchronous); default asyn

High-resolution graphics can be generated principally in two forms by Genstat: either on a screen that can operate in graphics mode or by sending output to a file. The screen-based operation is for use in interactive sessions, whereas file output is designed for later use outside Genstat: either to produce hard-copy on a plotter or laser-printer, or to re-display graphics on the screen, if appropriate software is available. Usually there is a choice of various kinds of screen type or file format. Each type of output, whether screen or file, is referred to as a *device*; thus, the first step in producing graphical output is selecting a device within Genstat that is appropriate for the hardware that you have available. Genstat has built-in interfaces to several different graphics devices. These vary according to the Genstat implementation. However, details of the devices, their characteristics and their associated numbers can be obtained from the DHELP procedure by typing the statement

```
DHELP possible
```

The output device is selected by the DEVICE statement. For example

```
DEVICE 4
```

selects the fourth available device.

If you have selected a file-based device you also need to open a file to receive the output, using the OPEN directive. This can be done before or after selecting the device, so long as the file has been opened before any output is generated. You can close the file when the graphics are complete; if you want to store separate items of graphical output in individual files you can use a sequence of `OPEN` and `CLOSE` statements (3.3). When opening or closing files for graphical output the `CHANNEL` parameter of the `OPEN` and `CLOSE` statements should be set to the device number specified by the `DEVICE` statement. For example:

```
OPEN 'Plot.jpg'; CHANNEL=7; FILETYPE=graphics
DEVICE 7
DGRAPH Y; X
CLOSE 7; FILETYPE=graphics
```

The default device, selected automatically when you start Genstat, is device 1: sometimes you may be able to specify an alternative device number and associated output file on the command line used to start Genstat (the local Genstat documentation should explain if this is possible).

You may get strange results if you try to generate graphics on a screen that is not designed for displaying graphics, or if you specify the wrong device type, as Genstat is not always able to detect the type of device or screen.

There need be little difference in your use of Genstat graphics on different devices as, by default, the plotting symbols and character output are *software-generated* using built-in graphics definitions that are supplied with Genstat. It may sometimes be advantageous, however, to use particular features of the device. These device-specific features are usually selected by negative parameter settings (for example, by setting parameter `SYMBOL=-3` in the `PEN` directive; 6.9.8). Naturally, selection of device-specific attributes may lead to some differences in appearance of the output on different devices. Likewise different devices may have different initial default settings, in particular according to whether or not they support colour. Details of these device-specific properties are provided in the information provided by the `DHELP` procedure, as explained above.

The `ENDACTION` parameter, with settings `continue` and `pause`, controls the action taken by default at the end of each plot. When using a graphics terminal interactively it may be convenient to pause at the end of a plot to examine the screen. When you are ready to continue, pressing carriage-return or some equivalent key will switch the terminal back to text mode and the Genstat prompt will appear. The `DHELP` statement above should provide the precise details for each particular device. For some interactive devices, for example PCs or workstations with separate graphics windows, it may not be necessary to pause. Each device is initialized to either `pause` or `continue` when you start Genstat, according to the particular implementation. If you are running in batch mode the default will always be to `continue`.

You can repeat the `DEVICE` statement and set `ENDACTION` to `pause` or `continue` at any time that you wish to change the default action. Alternatively, each graphical directive has an `ENDACTION` option that controls the device at the end of that directive, without altering the general default setting. For example, if you wish to build up a complex display using several `DGRAPH` statements with option `SCREEN=keep`, you could set `ENDACTION=continue` in the `DEVICE` statement, and then put `ENDACTION=pause` in the final `DGRAPH` statement.

The `ORIENTATION` parameter can be used to specify `landscape` or `portrait` orientation of graphical output on PostScript and Interacter raster devices; `portrait` is the default. `PALETTE` can be set to `monotone`, to force all colours to be mapped to colour 1; this is the default for PostScript. Alternatively, `PALETTE=colour` produces colour PostScript output. The additional setting `PALETTE=greyscale` is as for monotone except that area filling (as in histograms) are shaded in grey tones, using the red component of the colour to define the grey intensity.

The `RESOLUTION` parameter specifies the height of the image for hard-copy output, in pixels. (This is equivalent to setting the image resolution in the Options menu of the Genstat Graphics Viewer.)

The `ACTION` parameter controls how graphs are created for the file types `.emf`, `.jpg`, `.tif`,

`.png`, `.gmf` and `.bmp`. The setting `synchronous` creates the graph before executing another command, whereas the setting `asynchronous` allows subsequent commands to be executed whilst the graph is created.

6.9.2 Re-displaying the graphics screen

`DDISPLAY` directive

Redraws the current graphical display.

Options

`DEVICE` = *scalar*	Device on which to redraw the display (on some systems it may only be possible to redisplay the picture on an interactive graphics device); default uses the current graphics device
`ENDACTION` = *string token*	Action to be taken after completing the plot (`continue`, `pause`); default `*` uses the setting from the last `DEVICE` statement

No parameters

This directive is provided to allow additional control of some interactive devices. In some of these the screen can operate in either text mode or graphics mode. Genstat will automatically switch the screen into the appropriate mode when starting or finishing a graph. Having returned to text mode after examining a graph you may later wish to have another look at the graph that was plotted. `DDISPLAY` will switch the screen back to graphics mode, thus re-displaying the graph. The `ENDACTION` option controls what happens after re-displaying the graph; normally with this type of device you would want to pause. The default action for `DDISPLAY` is the setting specified by the most recent `DEVICE` statement.

This directive has no effect when output is directed to a graphics file. For devices that do not operate in this dual-mode fashion, for example a graphics window under X-windows, `DDISPLAY` has no effect on the graphical display itself. It will however generate a pause if `ENDACTION` is set to request one.

Note that `DDISPLAY` does not actually re-plot the graphical output; it merely switches the screen into graphics mode, and assumes that your system has preserved the graphics image.

6.9.3 The `FRAME` directive

`FRAME` directive

Defines the positions and appearance of the plotting windows within the frame of a high-resolution graph.

Options

`GRID` = *string tokens*	Specifies grid lines (`xy`, `xz`, `yx`, `yz`, `zx`, `zy`)
`BOXFRAME` = *string tokens*	Whether to include a box enclosing the entire frame (`include`, `omit`)
`BACKGROUND` = *scalars* or *texts*	Specifies the colour to be used for the background of the whole frame (where allowed by the graphics device)

`RESET` = *string token*	Whether to reset the window definition to the default values (`no`, `yes`); default `no`

Parameters

`WINDOW` = *scalars*	Window numbers
`YLOWER` = *scalars*	Lower y device coordinate for each window
`YUPPER` = *scalars*	Upper y device coordinate for each window
`XLOWER` = *scalars*	Lower x device coordinate for each window
`XUPPER` = *scalars*	Upper x device coordinate for each window
`YMLOWER` = *scalars*	Size of bottom margin (for x-axis labels)
`YMUPPER` = *scalars*	Size of upper margin (for overall title)
`XMLOWER` = *scalars*	Size of left-hand margin (for y-axis labels)
`XMUPPER` = *scalars*	Size of right-hand margin
`BACKGROUND` = *scalars* or *texts*	Specifies the colour to be used for the background in each window (where allowed by the graphics device)
`BOX` = *string tokens*	Whether to include a box enclosing the plotted graphic (`include`, `omit`)
`BOXSURFACE` = *string tokens*	Box to include in a surface plot (`full`, `bounded`, `omit`)
`BOXKEY` = *string tokens*	Box to draw around key (`full`, `bounded`, `omit`)
`PENTITLE` = *scalars*	Pen to use to write the overall title
`PENKEY` = *scalars*	Pen to use for the key
`PENGRID` = *scalars*	Pen to use to draw the grid lines
`SCALING` = *string tokens*	How to scale the axis in each window (`xyequal`, `xzequal`, `yzequal`, `xyzequal`)
`TPOSITION` = *string tokens*	Position of title (`right`, `left`, `center`, `centre`)
`CINTERIOR` = *scalars* or *texts*	Specifies the colour to be used for the interior of each window (where allowed by the graphics device)
`CFRAME` = *scalars* or *texts*	Specifies the colour to be used for the frame of each window (where allowed by the graphics device)
`CTITLE` = *scalars* or *texts*	Specifies the colour to be used for the title bar of each window (where allowed by the graphics device)
`AXES` = *identifiers* or *pointers*	Additional oblique axes to include in each window
`SAVE` = *pointers*	Saves details of the current settings for the window concerned

You can define up to 256 different windows in which to plot graphics. Each window is a rectangular area of the screen which is defined using *normalized device coordinates* (NDC). For all devices, you can assume that a range of 0.0 to 1.0 will be available in both y- and x-directions, thus defining a 1 × 1 square to represent the plotting area. On some devices the plotting area may extend further in either the y- or x-direction (but not both). Details can be obtained using the `DHELP` procedure, as explained at the start of Section 6.9. By keeping within the [0,1] range you can ensure that the window is always valid, whatever output device is selected. However, you may wish to use the extended area where possible on a particular device.

The mapping from NDC to physical coordinates on the current output device is performed internally, so the window definitions are independent of the actual size of device. The NDC coordinates are also completely independent of the values of the data that are to be plotted. (The locations of the points within the graph depend on how the axes of the graph are defined; see 6.9.4, 6.9.5 and 6.9.6).

When you use `FRAME`, any aspects of the windows that you do not specify explicitly retain the values that they had immediately before the `FRAME` statement. Alternatively, you can specify option `RESET=yes` to reset all these aspects to the default values, defined by Genstat at the start of each job.

To define a window, the upper and lower bounds are required in both y- and x-directions; thus defining both the position and the size of the window. For example

```
FRAME WINDOW=1; YLOWER=0.25; YUPPER=0.75;\
      XLOWER=0; XUPPER=0.5
```

defines window 1 to be a square of size 0.5, whose bottom left corner is at the point (0.0,0.25), and whose top right corner is (0.5,0.75). This does not define the exact size of a graph plotted in this window, as margins may be required for the annotation and titles (see below).

If you do not specify all four values in the `FRAME` statement, the existing values are retained. A check is then made on the validity of the window bounds. The settings of `YLOWER` and `XLOWER` must be strictly less than those of `YUPPER` and `XUPPER` respectively; also, none of the bounds can be outside the permitted range, which is [0.0,1.0] on most graphics devices. You cannot use `*` to reset a bound to the default value; if you try to do so, Genstat will produce an error diagnostic. (Instead you can specify option `RESET=yes`, as explained above.)

All the windows have a default size defined when you start Genstat. Window 1 is the default window used for plots by `DGRAPH`, `DCONTOUR`, and so on, and is set up to be a square of size 0.75. The default key window is window 2, which is a rectangle of height 0.25 and width 0.75 located immediately below window 1. Windows 3 and 4 are the unit square [0,1]×[0,1] and windows 5, 6, 7 and 8 are the top-left, top-right, bottom-left, and bottom-right quarters respectively of the unit square. Windows 9, 10, 11 and 12 also divide the frame into quarters, but they have the full width (0 to 1) in the x-direction and quarter of the width in the y-direction, working from the top (i.e. 0.75 to 1 for window 9) to the bottom (i.e. 0 to 0.25 for window 12) of the frame. The remaining windows, from 13 to 256, also default to the unit square. You can use `FRAME` to modify the size or position of any of these windows.

Usually, a margin is provided around each plot so that there is room for the axes to be drawn, along with labelling and titles as specified by the `XAXIS` or `YAXIS` directives (6.9.4 and 6.9.5). By default, the margin size is designed to allow sufficient room for annotation to be added using the standard character size, as defined by the `SIZEMULTIPLIER` or `SMLABEL` parameters of `PEN` (6.9.8). If you use `XAXIS` or `YAXIS` to control the plotting of axes explicitly you may wish to alter the size of the margins, either to increase the space used for the axes or, alternatively, to maximize the space available for the graph itself. For example, if you alter the size of the labelling, by explicitly defining the relevant axis pens, more space may be required for the axes; otherwise the labels may be clipped at the window bounds. The parameters `YMLOWER`, `YMUPPER`, `XMLOWER` and `XMUPPER` can be used to set the space (in NDC) for the bottom, top, left-hand and right-hand margins respectively, and have initial default settings of 0.10, 0.07, 0.12 and 0.05.

On most devices the background colours of the window may be modified by setting the `BACKGROUND`, `CINTERIOR`, `CFRAME` and `CTITLE` parameters. The `BACKGROUND` parameter can be used to define the colour for the whole background, while the other parameters define specific aspects (overriding any setting of `BACKGROUND`): `CINTERIOR` defines the colour of the interior of the plot (where the points are plotted), `CFRAME` defines the colour of the outer frame (outside the interior), and `CTITLE` is the colour of the title bar. The parameters can be set either to a text containing the name of one of Genstat's pre-defined colours (6.9.9), or to a scalar containing a number defining a colour using the RGB system; see 6.9.9. Similarly, the `BACKGROUND` option can define the background colour for the whole frame (which may include areas outside any of the windows). The special colour setting `'match'` can be used to apply the colour from the preceding parameter to the next one: `CFRAME` would inherit the colour from `CINTERIOR`, and `CTITLE` would inherit from from `CFRAME`. For example,

```
FRAME 1; CINTERIOR='white'; CFRAME='ivory'; CTITLE='match'
```

will specify colour white for the inside of the plot, and ivory to all the area outside this.

The `PENTITLE` and `PENKEY` options allow you to define the pens to be used to write the overall title and the key in each window; the initial default is to use pen -5 and -6 respectively.

The TPOSITION parameter can be used to specify the position of the title in each window: either left-justified, right-justified or centred. The initial default is that it is centred.

The GRID option allows you to request grid lines to be drawn in particular directions and planes (for all the windows listed by the WINDOW parameter). For example the setting xy requests lines in the xy plane running from the x-axis (that is, parallel to the y-axis), and the setting yx requests lines in the xy plane running from the y-axis (that is, parallel to the x-axis); so you can set both of these to obtain box markings in the xy plane. The PENGRID option specifies the pen to be used for the grid lines in each window; the initial default is to use pen -4. You must use the RESET option if you want to restore these pen numbers to the initial defaults. (Genstat does not allow you to set negative pen numbers explicitly.) The BOX parameter allows you to put a box around the window in plots other than surface plots; the initial default is to omit this. The box for a surface plot is controlled by the BOXSURFACE option, and can either be a full box enclosing the whole graph, or a bounded box enclosing just the surface; the initial default is that no box is drawn. The BOXKEY parameter can request that either a full or a bounded box be drawn around each key; the initial default is to omit the box. Finally, the BOXFRAME option controls whether or not a box is drawn around the entire frame; the initial default is to include the box.

The SCALING parameter enables you to request that scaling of the x-, y- or z-axes should be equal in each window. For example, the xyequal setting ensures that the x- and y-axes are scaled identically, the setting xyzequal ensures that all the axes have the same scaling, and so on.

The AXES parameter allows you to specify the identifier of an oblique axis (defined by the AXIS directive; see 6.9.7) that should be included in a window. If you want to include several axes, you can specify a pointer containing the identifiers of the required axes.

The current FRAME settings for a particular window can be saved in a pointer supplied by the SAVE parameter. The elements of the pointer are labelled to identify the components, as shown in Example 6.9.3.

Example 6.9.3

```
   2  FRAME 1; YLOWER=0.0; XUPPER=1.0; SAVE=Win1
   3  PRINT [ORIENTATION=across; RLWIDTH=18] Win1[]

       Win1['grid']
   Win1['boxframe']        omit
Win1['fbackground']           *
     Win1['ylower']           0
     Win1['yupper']       1.000
     Win1['xlower']           0
     Win1['xupper']       1.000
    Win1['ymlower']      0.1000
    Win1['ymupper']     0.07000
    Win1['xmlower']      0.1200
    Win1['xmupper']     0.05000
 Win1['background']          -1
        Win1['box']     include
 Win1['boxsurface']        full
     Win1['boxkey']     bounded
   Win1['pentitle']          -5
     Win1['penkey']          -6
    Win1['pengrid']          -4
    Win1['scaling']
  Win1['tposition']      centre
  Win1['cinterior']          -1
     Win1['cframe']          -1
     Win1['ctitle']          -1
```

An alternative to FRAME, for setting up a plot-matrix of windows, is to use the FFRAME procedure. This uses FRAME internally to define windows in either a rectangular, square, `lowersymmetric`, `uppersymmetric` or `diagonal` pattern.

6.9.4 The `XAXIS` directive

There is a definition for the axes associated with each Genstat graphics window, which specifies how the axes are to be drawn when graphical output is produced in that window. The default definition for each set of axes requires some of the features to be determined from the data, as described below. Others have fixed defaults that are independent of the data. The `XAXIS` directive can be used to override the default action and specify particular aspects of the x-axis explicitly. Similarly, directives `YAXIS` (6.9.5) and `ZAXIS` (6.9.6) modify the y- and z-axis definitions, respectively.

`XAXIS` directive

Defines the x-axis in each window for high-resolution graphics.

Option

`RESET` = *string token*	Whether to reset the axis definition to the default values (`no`, `yes`); default `no`

Parameters

`WINDOW` = *scalars*	Numbers of the windows
`TITLE` = *texts*	Title for the axis
`TPOSITION` = *string tokens*	Position of title (`middle`, `end`)
`TDIRECTION` = *string tokens*	Direction of title (`parallel`, `perpendicular`)
`LOWER` = *scalars*	Lower bound for axis
`UPPER` = *scalars*	Upper bound for axis
`MARKS` = *scalars* or *variates*	Distance between each tick mark (scalar) or positions of the marks along the axis (variate)
`MPOSITION` = *string tokens*	Positioning of the tick marks on the axis (`inside`, `outside`, `across`)
`LABELS` = *texts* or *variates*	Labels at each major tick mark
`LPOSITION` = *string tokens*	Position of the axis labels (`inside`, `outside`)
`LDIRECTION` = *string tokens*	Direction of the axis labels (`parallel`, `perpendicular`)
`LROTATION` = *scalars* or *variates*	Rotation of the axis labels
`YORIGIN` = *scalars*	Position on y-axis at which the axis is drawn
`ZORIGIN` = *scalars*	Position on z-axis at which the axis is drawn
`PENTITLE` = *scalars*	Pen to use to write the axis title
`PENAXIS` = *scalars*	Pen to use to draw the axis
`PENLABELS` = *scalars*	Pen to use to write the axis labels
`ARROWHEAD` = *string tokens*	Whether the axis should have an arrowhead (`include`, `omit`)
`ACTION` = *string tokens*	Whether to display or hide the axis (`display`, `hide`)
`TRANSFORM` = *string tokens*	Transformed scale for the axis (`identity`, `log`, `log10`, `logit`, `probit`, `cloglog`, `square`, `exp`, `exp10`, `ilogit`, `iprobit`, `icloglog`, `root`); default `iden`
`LINKED` = *scalars*	Linked axis whose definitions should be used for this axis in a 2-dimensional graph; default `*` i.e. none
`MLOWER%` = *scalars*	How large a margin to set between the lowest x-value and the lower value of the axis, if not set explicitly by `LOWER` (expressed as a percentage of the range of the x-values)

MUPPER% = *scalars*	How large a margin to set between the largest x-value and the upper value of the axis, if not set explicitly by UPPER (expressed as a percentage of the range of the x-values)
DECIMALS = *scalars* or *variates*	Number of decimal places to use for numbers printed at the marks
DREPRESENTATION = *scalars*, *variates* or *texts*	Format to use for dates and times printed at the marks
VREPRESENTATION = *string tokens*	Format to use for numbers printed at the marks (decimal, engineering, scientific); default deci
YOMETHOD = *string tokens*	Method to use to set the position of the origin on the y-axis if not set explicitly by YORIGIN (upper, lower, center, centre)
ZOMETHOD = *string tokens*	Method to use to set the position of the origin on the z-axis if not set explicitly by ZORIGIN (upper, lower, center, centre)
REVERSE = *string tokens*	Whether to reverse the axis direction to run from upper to lower instead of the default lower to upper (yes, no); default no
SAVE = *pointers*	Saves details of the current settings for the axis concerned

All the parameters of XAXIS are relevant when using DGRAPH (6.2.1), but for other directives only some of the parameters are used.

The WINDOW parameter specifies the window whose axis definition is to be altered. WINDOW can be set to a list of window numbers, in which case the other parameter lists are cycled in parallel, in the usual way. By default, only those aspects specified by subsequent parameter lists are modified; any parameters that are not set will retain their current settings. Alternatively, you can specify option RESET=yes to reset the values of any parameters that are not set for each window, back to the default values that are set up by Genstat at the start of a job.

The LOWER and UPPER parameters specify the lower and upper bounds for the axis. By default, Genstat derives suitable axis bounds from the data, as described for the appropriate directive. You can obtain an inverted scale by setting parameter REVERSE=yes. The values specified with these parameters are on the scale of the data values that are plotted, and are independent of the normalized device coordinates used to define the window size in FRAME (6.9.3). The MLOWER% parameter controls the size of margin that is provided between the lower value of the axis and the smallest x-value, if the lower axis value is not set explicitly by LOWER. This is expressed as a percentage of the range of the x-values, and has the initial default of 5%. Similarly the MUPPER% parameter controls the size of the upper margin.

The YORIGIN parameter determines the value on the y-axis through which the axis is drawn. If its value is outside the y-axis bounds, the upper or lower bound is adjusted so that the axis will extend up to the specified origin. This applies whether you have set the bounds explicitly or have left Genstat to calculate them from the data. If YORIGIN is not set, the YOMETHOD parameter can specify how the position should be determined: either at the upper value on the y-axis, or the lower value, or in the centre. The initial default (if neither of these parameters has been specified) is to put the axis at the bottom of the y-axis, which will be the lower value unless the scale is reversed. The ZORIGIN and ZOMETHOD parameters set the position of the origin on the z-axis in a similar way.

You can specify a title for the axis using the TITLE parameter. This is limited to a single line of characters. The TPOSITION parameter controls whether the title is placed in the middle or at the end of the axis, and the TDIRECTION parameter controls whether it is written parallel or perpendicular to the axis.

The axis is marked with a scale, determined automatically so that tick marks are evenly spaced and positioned to give "round" numbers for the scale values. You can set the `MARKS` parameter to a scalar to define the increment between tick marks. For example, setting `MARKS=1.5` with bounds 10 and 2 causes tick marks to appear at 2, 3.5, 5, 6.5, 8 and 9.5. The interval must be a positive number, irrespective of the values of the bounds. Alternatively, you can set `MARKS` to a variate (with more than one value) to specify the actual positions of the tick marks on the axis. Any values that lie outside the axis bounds are ignored. The scale values printed next to the tick marks use a format that is determined automatically from the values, but if you set `MARKS` to a variate it will use the number of decimals specified in the variate declaration. If `MARKS` is unset or set to a scalar, you can use the `NSUBTICKS` parameter to specify a number of "subticks" to be drawn between each of the (major) tick marks.

When you set `MARKS`, you can also use the `LABELS` parameter to specify a set of labels to print at the (major) axis marks, instead of the numbers. For example,

```
    TEXT [VALUES=Mon,Tues,Wed,Thur,Fri,Sat,Sun] Day
    VARIATE [VALUES=1...31] Month
    XAXIS 1; MARKS=Month; LABELS=Day
```

The strings within the text are cycled if necessary, so the number of strings can be less than the number of tick marks. The `DECIMALS` parameter can set the number of decimal places to use if you are printing numbers at the marks. If the numbers represent dates or times, you should specify their formats using the `DREPRESENTATION` parameter (see 2.1.5). By default, numbers are printed in decimal form. If you would prefer scientific format you can set parameter `VREPRESENTATION=scientific`; numbers are then printed as a decimal number with absolute value less than 10, followed by an exponent (e.g. 3.4567 E4 for 34567). Alternatively, you can set `VREPRESENTATION=engineering` to use engineering format; the decimal number then has an absolute value less than 10000, so the exponent is a multiple of 3 (e.g. 34.567 E3 for 34567). With scientific or engineering formats, the `DECIMALS` parameter sets the number of significant figures to use rather than the number of decimal places.

The `MPOSITION` parameter controls the positioning of the tick marks, which can be drawn on the inside or the outside of the axis, or can be drawn across the axis. With the `outside` setting, the tick marks are drawn towards the outside of the plot; that is below the axis if the axis is in the lower half of the plot, or above the axis if it is in the top half of the plot. The aim is then to position the tick marks away from the main part of the plot, so that they interfere with the plotted points as little as possible. With the `inside` setting, the marks are drawn on the opposite side (that is, to the inside of the plot), while the `across` setting draws them across the axis. Similarly, the positioning of the scale markings or labels is controlled by the `LPOSITION` parameter, with settings `inside` or `outside`. The `LDIRECTION` parameter controls whether the scale markings or labels are written parallel or perpendicular to the axis. Alternatively, you can use the `LROTATION` parameter to specify the direction of the labels more precisely, as a rotation in degrees from the horizontal (i.e. parallel) direction. If `LROTATION` is specified, any setting of `LDIRECTION` is ignored.

Setting `MARKS=*` will return to the default positioning of the tick marks. Setting `LABELS=*` will switch off any labels previously specified. Setting `MPOSITION=*` will switch off any tick marks, and setting `LPOSITION=*` or `LDIRECTION=*` will switch off any labels.

The `TRANSFORM` parameter allows you to transform the scale of the axis. The tick marks are still defined and labelled according to the original scale, but their physical positions on the graph are transformed. So, for example, with `TRANSFORM=log10`, the equal physical distance between 1 and 10 would be the same as the distance between 10 and 100. The settings are the same as the names of the equivalent Genstat functions, with the addition of `exp10` for the antilog transformation (i.e. 10^x), `and square` for x^2.

There are three parameters to control the pens to be used to draw the axis. These are

PENTITLE, PENAXIS and PENLABEL, specifying the pen for the title, the axis and the labelling, respectively. The initial default is to use pens -1, -2 and -3 in every window. These pens are given negative numbers to allow them to be distinguished from the pens used for the contents of the plot. They are initially set up to use colour black, line style 1, thickness 1 and size 1. You can thus control which pens are used for drawing the axis in each window, and the attributes of those pens. For example, if no XAXIS statement has yet been given,

```
PEN -1; LINESTYLE=4; COLOUR=2
```

will request that the titles in every window should be written in line style 4 and colour 2; while

```
PEN 29; LINESTYLE=3; COLOUR=4
XAXIS 1; PENAXIS=29
```

will change the appearance of just the x-axis in window 1, as pen 29 is not used for the other windows. You should of course be careful of side-effects when changing the pen numbers. For example, pen 29 may also have been modified for use in a DGRAPH statement and other attributes may have been set that are not wanted when drawing the axis. You must use the RESET option if you want to restore these pen numbers to the initial defaults. (Genstat does not allow you to set negative pen numbers explicitly.)

The ARROWHEAD parameter controls whether the axis is drawn with an arrowhead at the end.

You may sometimes wish to use the axis definitions merely to control the positioning of the plot in the x-direction (using the UPPER and LOWER parameters), or you may wish to hide the axis temporarily in case it is obscuring information in the plot. You can do this by setting parameter ACTION=hide.

Axis annotation is plotted in the margins specified by the FRAME directive (6.9.3). You may wish to reduce the size of these margins if you have defined axes that use less space, for example by keeping within the area of the graph itself, or by omitting titles or labels. Space can thus be regained and used for plotting data. However, if the margins are too small the axis annotation may be "clipped" at the boundaries of the margins; if this happens, you can use FRAME to increase the margin size. The margins are used by DGRAPH (6.2.1), DHISTOGRAM (6.3.1) and DCONTOUR (6.4.1), but they are ignored by other directives.

The LINKED parameter is useful when you have several related plots in different windows within the frame. If, for example, you set LINKED=n, the attributes of the current x-axis will all be taken (at the time of plotting) from the definition of the x-axis for any 2-dimensional graph in window *n*. Also, you can edit the attributes of all the linked axes simultaneously in the graphics viewer in Genstat *for Windows*.

The current settings of the axis for a particular window can be saved in a pointer supplied by the SAVE parameter. The SAVE parameter The elements of the pointer are labelled to identify the components, as shown in Example 6.9.4.

Example 6.9.4

```
  2  XAXIS 7; TITLE='x-axis'; LOWER=2; UPPER=10; MARKS=1.5; SAVE=Axes7
  3  PRINT [ORIENTATION=across; RLWIDTH=19] Axes7[]

          Axes7['title']        x-axis
      Axes7['tposition']        middle
     Axes7['tdirection']      parallel
          Axes7['lower']          2.00
          Axes7['upper']         10.00
          Axes7['marks']         1.500
      Axes7['mposition']       outside
         Axes7['labels']
      Axes7['lposition']       outside
     Axes7['ldirection']      parallel
      Axes7['lrotation']             *
      Axes7['nsubticks']             0
        Axes7['yorigin']             *
        Axes7['zorigin']             *
       Axes7['pentitle']          -1
        Axes7['penaxis']          -2
      Axes7['penlabels']          -3
      Axes7['arrowhead']        omit
         Axes7['action']     display
      Axes7['transform']    identity
         Axes7['linked']           *
        Axes7['mlower%']       5.000
        Axes7['mupper%']       5.000
       Axes7['decimals']           *
Axes7['drepresentation']           *
Axes7['vrepresentation']    decimals
       Axes7['yomethod']           *
       Axes7['zomethod']           *
        Axes7['reverse']          no
```

The settings are those for the axis itself, so you should check that the axis is not linked to one in another window. (The `'linked'` element contains the window number, or a missing value there is no link.) This facility is of most use within procedures, where you may wish to check or modify particular axis settings before constructing complicated graphs. Also, the `DKEEP` directive (6.9.10) allows you to extract the actual bounds used when plotting; these will be the bounds determined from the data if none have been defined explicitly by `XAXIS`.

6.9.5 The `YAXIS` directive

`YAXIS` directive

Defines the y-axis in each window for high-resolution graphics.

Option

`RESET` = *string token*	Whether to reset the axis definition to the default values (`no`, `yes`); default `no`

Parameters

`WINDOW` = *scalars*	Numbers of the windows
`TITLE` = *texts*	Title for the axis
`TPOSITION` = *string tokens*	Position of title (`middle`, `end`)
`TDIRECTION` = *string tokens*	Direction of title (`parallel`, `perpendicular`)
`LOWER` = *scalars*	Lower bound for axis
`UPPER` = *scalars*	Upper bound for axis
`MARKS` = *scalars* or *variates*	Distance between each tick mark (scalar) or positions of the marks along the axis (variate)
`MPOSITION` = *string tokens*	Positioning of the tick marks on the axis (`inside`, `outside`, `across`)
`LABELS` = *texts* or *variates*	Labels at each major tick mark
`LPOSITION` = *string tokens*	Position of the axis labels (`inside`, `outside`)
`LDIRECTION` = *string tokens*	Direction of the axis labels (`parallel`, `perpendicular`)
`LROTATION` = *scalars* or *variates*	Rotation of the axis labels
`NSUBTICKS` = *scalars*	Number of subticks per interval (ignored if `MARKS` is a variate)
`XORIGIN` = *scalars*	Position on x-axis at which the axis is drawn
`ZORIGIN` = *scalars*	Position on z-axis at which the axis is drawn
`PENTITLE` = *scalars*	Pen to use to write the axis title
`PENAXIS` = *scalars*	Pen to use to draw the axis

PENLABELS = *scalars*	Pen to use to write the axis labels
ARROWHEAD = *string tokens*	Whether the axis should have an arrowhead (include, omit)
ACTION = *string tokens*	Whether to display or hide the axis (display, hide)
TRANSFORM = *string tokens*	Transformed scale for the axis (identity, log, log10, logit, probit, cloglog, square, exp, exp10, ilogit, iprobit, icloglog, root); default iden
LINKED = *scalars*	Linked axis whose definitions should be used for this axis in a 2-dimensional graph; default * i.e. none
MLOWER% = *scalars*	How large a margin to set between the lowest y-value and the lower value of the axis, if not set explicitly by LOWER (expressed as a percentage of the range of the y-values)
MUPPER% = *scalars*	How large a margin to set between the largest y-value and the upper value of the axis, if not set explicitly by UPPER (expressed as a percentage of the range of the y-values)
DECIMALS = *scalars* or *variates*	Number of decimal places to use for numbers printed at the marks
DREPRESENTATION = *scalars*, *variates* or *texts*	Format to use for dates and times printed at the marks
VREPRESENTATION = *string tokens*	Format to use for numbers printed at the marks (decimal, engineering, scientific); default deci
XOMETHOD = *string tokens*	Method to use to set the position of the origin on the x-axis if not set explicitly by XORIGIN (upper, lower, center, centre)
ZOMETHOD = *string tokens*	Method to use to set the position of the origin on the z-axis if not set explicitly by ZORIGIN (upper, lower, center, centre)
REVERSE = *string tokens*	Whether to reverse the axis direction to run from upper to lower instead of the default lower to upper (yes, no); default no
SAVE = *pointers*	Saves details of the current settings for the axis concerned

The syntax of YAXIS is identical to that of XAXIS (6.9.4), except that YAXIS has XORIGIN and XOMETHOD parameters which replaces the YORIGIN and YOMETHOD parameters of XAXIS. All the parameters are relevant when using DGRAPH (6.2.1), but for other directives only some of the parameters are used.

As in XAXIS, the WINDOW parameter specifies the window whose axis definition is to be altered. By default, only those aspects specified by subsequent parameter lists are modified, but you can specify option RESET=yes to reset the values of any parameters that are not set for each window, back to the default values that are set up by Genstat at the start of a job. The LOWER, UPPER, MLOWER% and MUPPER% parameters again specify the lower and upper bounds for the axis, the REVERSE parameter can reverse the axis, and the TITLE, TPOSITION and TDIRECTION parameter can define a title for the axis.

The XORIGIN parameter determines the value on the x-axis through which the axis is drawn. If its value is outside the x-axis bounds, the upper or lower bound is adjusted so that the axis will extend up to the specified origin. This applies whether you have set the bounds explicitly or have left Genstat to calculate them from the data. If XORIGIN is not set, the XOMETHOD parameter can specify how the position should be determined: either at the upper value on the x-axis, or the lower value, or in the centre. The initial default (if neither of these parameters has been specified) is to put the axis at the left-hand end of the x-axis, which will be the lower value unless the scale is reversed. The ZORIGIN and ZOMETHOD parameters set the position of the origin on the z-axis in a similar way, with the initial default that the axis is at the bottom of the z-axis.

The MARKS, NSUBTICKS, LABELS, DECIMALS, DREPRESENTATION and VREPRESENTATION parameters also operate as in XAXIS, to specify the markings on the axis, and their associated labels. The MPOSITION, LPOSITION, LDIRECTION and LROTATION parameters again control the positioning of the tick marks and labels. For a y-axis, the outside setting implies that the tick marks are drawn to the left of the axis if the axis is on the left-half side of the plot, or to the right of the axis if it is on the right-hand side. As in XAXIS, the TRANSFORM parameter allows you to transform the physical scale of the axis on the graph.

The ARROWHEAD parameter again controls whether the axis is drawn with an arrowhead at the end, and parameters PENTITLE, PENAXIS and PENLABEL specify the to be used for the title, the axis and the labelling, respectively. ACTION allows you to hide the axis, LINKED allows you to take all the axis settings from a (linked) axis in another window, and SAVE allows you to save the current settings defined for the axis. Further details are given in the description of XAXIS (6.9.4).

6.9.6 The ZAXIS directive

ZAXIS directive

Defines the z-axis in each window for high-resolution graphics.

Option

RESET = *string token*	Whether to reset the axis definition to the default values (no, yes); default no

Parameters

WINDOW = *scalars*	Numbers of the windows
TITLE = *texts*	Title for the axis
TPOSITION = *string tokens*	Position of title (middle, end)
TDIRECTION = *string tokens*	Direction of title (parallel, perpendicular)
LOWER = *scalars*	Lower bound for axis
UPPER = *scalars*	Upper bound for axis
MARKS = *scalars* or *variates*	Distance between each tick mark (scalar) or positions of the marks along the axis (variate)
MPOSITION = *string tokens*	Positioning of the tick marks on the axis (inside, outside, across)
LABELS = *texts* or *variates*	Labels at each major tick mark
LPOSITION = *string tokens*	Position of the axis labels (inside, outside)
LDIRECTION = *string tokens*	Direction of the axis labels (parallel, perpendicular)
LROTATION = *scalars* or *variates*	Rotation of the axis labels
NSUBTICKS = *scalars*	Number of subticks per interval (ignored if MARKS is a variate)
XORIGIN = *scalars*	Position on x-axis at which the axis is drawn
YORIGIN = *scalars*	Position on y-axis at which the axis is drawn
PENTITLE = *scalars*	Pen to use to write the axis title
PENAXIS = *scalars*	Pen to use to draw the axis
PENLABELS = *scalars*	Pen to use to write the axis labels

`ARROWHEAD` = *string tokens*	Whether the axis should have an arrowhead (`include`, `omit`)
`ACTION` = *string tokens*	Whether to display or hide the axis (`display`, `hide`)
`MLOWER%` = *scalars*	How large a margin to set between the lowest z-value and the lower value of the axis, if not set explicitly by `LOWER` (expressed as a percentage of the range of the z-values)
`MUPPER%` = *scalars*	How large a margin to set between the largest z-value and the upper value of the axis, if not set explicitly by `UPPER` (expressed as a percentage of the range of the z-values)
`DECIMALS` = *scalars* or *variates*	Number of decimal places to use for numbers printed at the marks
`DREPRESENTATION` = *scalars*, *variates* or *texts*	Format to use for dates and times printed at the marks
`VREPRESENTATION` = *string tokens*	Format to use for numbers printed at the marks (`decimal`, `engineering`, `scientific`); default `deci`
`XOMETHOD` = *string tokens*	Method to use to set the position of the origin on the x-axis if not set explicitly by `XORIGIN` (`upper`, `lower`, `center`, `centre`)
`YOMETHOD` = *string tokens*	Method to use to set the position of the origin on the y-axis if not set explicitly by `YORIGIN` (`upper`, `lower`, `center`, `centre`)
`REVERSE` = *string tokens*	Whether to reverse the axis direction to run from upper to lower instead of the default lower to upper (`yes`, `no`); default `no`
`SAVE` = *pointers*	Saves details of the current settings for the axis concerned

The syntax of `ZAXIS` is identical to that of `XAXIS` (6.9.4), except that `ZAXIS` has an `XORIGIN` parameter instead of the `ZORIGIN` parameter of `XAXIS`. All parameters are relevant when using `D3GRAPH` (6.2.2), but for other directives only some of the parameters are used.

The `XORIGIN` parameter determines the value on the x-axis through which the axis is drawn. If its value is outside the x-axis bounds, the upper or lower bound is adjusted so that the axis will extend up to the specified origin. This applies whether you have set the bounds explicitly or have left Genstat to calculate them from the data. If `XORIGIN` is not set, the `XOMETHOD` parameter can specify how the position should be determined: either at the upper value on the x-axis, or the lower value, or in the centre. The initial default (if neither of these parameters has been specified) is to put the axis at the left-hand end, which will be the lower value unless the scale is reversed.

6.9.7 The `AXIS` directive

`AXIS` directive

Defines an oblique axis for high-resolution graphics.

Option

`RESET` = *string token*	Whether to reset the axis definition to the default values (`yes`, `no`); default `no`

Parameters

IDENTIFIER = *identifiers*	Name to be used inside Genstat to identify each axis
TITLE = *texts*	Title for each axis
TPOSITION = *string tokens*	Position of title (middle, end)
TDIRECTION = *string tokens*	Direction of title (parallel, perpendicular)
LOWER = *scalars*	Lower bound for each axis
UPPER = *scalars*	Upper bound for each axis
MARKS = *scalars* or *variates*	Distance between each tick mark (scalar) or positions of the marks along each axis (variate)
MPOSITION = *string tokens*	Positioning of the tick marks on each axis (inside, outside, across)
LABELS = *texts* or *variates*	Labels at each major tick mark
LPOSITION = *string tokens*	Position of the axis labels (inside, outside)
LDIRECTION = *string tokens*	Direction of the axis labels (parallel, perpendicular)
LROTATION = *scalars* or *variates*	Rotation of the axis labels
NSUBTICKS = *scalars*	Number of subticks per interval (ignored if MARKS is a variate)
XZERO = *scalars*	Position of the axis origin in the x-dimension
YZERO = *scalars*	Position of the axis origin in the y-dimension
ZZERO = *scalars*	Position of the axis origin in the z-dimension
XSTEP = *scalars*	Step in the x-direction corresponding to a step of length one along the axis
YSTEP = *scalars*	Step in the y-direction corresponding to a step of length one along the axis
ZSTEP = *scalars*	Step in the z-direction corresponding to a step of length one along the axis
PENTITLE = *scalars*	Pen to use to write the axis title
PENAXIS = *scalars*	Pen to use to draw the axis
PENLABELS = *scalars*	Pen to use to write the axis labels
ARROWHEAD = *string tokens*	Whether the axis should have an arrowhead (include, omit)
ACTION = *string tokens*	Whether to display or hide the axis (display, hide)
TRANSFORM = *string tokens*	Transformed scale for the axis marks and labels (identity, log, log10, logit, probit, cloglog, square, exp, exp10, ilogit, iprobit, icloglog, root); default iden
DECIMALS = *scalars* or *variates*	Number of decimal places to use for numbers printed at the marks
DREPRESENTATION = *scalars* or *variates*	Format to use for dates and times printed at the marks
VREPRESENTATION = *string tokens*	Format to use for numbers printed at the marks (decimal, engineering, scientific); default deci
SAVE = *pointers*	Saves details of the current settings for the axis concerned

The AXIS directive allows you to define an oblique axis for high-resolution graphics. You use the IDENTIFIER parameter to supply an identifier to store the axis definition. You can then use this as a setting of the AXES parameter of the FRAME directive (6.9.3) to display the axis in a particular graphics window. The position of the axis origin in the x-, y- and z-dimensions of the window is specified by the parameters `XZERO`, `YZERO` and `ZZERO`, respectively. The `XSTEP`, `YSTEP` and `ZSTEP` parameters define the size of the steps in the x-, y- and z-directions that corresponds to a step of length one along the axis. These six parameters thus define the location and direction of the axis.

The `TRANSFORM` parameter allows you to transform the marks and labels on the axis. The location and direction of the axis are defined according to the original scale, by the `XZERO`, `YZERO`, `ZZERO`, `XSTEP`, `YSTEP` and `ZSTEP` parameters, as usual. The coordinates along the axis are then transformed, and labelled according to the transformed scale. So, for example, with `TRANSFORM=log10`, the original coordinates 1, 10 and 100 would be labelled 0, 1 and 2. The settings are the same as the names of the equivalent Genstat functions, with the addition of `exp10` for the antilog transformation (i.e. 10^x), `and square` for x^2.

The other parameters operate as in the `XAXIS` directive (6.9.4).

6.9.8 The `PEN` directive

`PEN` directive

Defines the properties of "pens" for high-resolution graphics.

Options

`RESET` = *string token*	Whether to reset the pen definitions to their default values (`yes`, `no`); default `no`
`BOXUNITS` = *string token*	Units to use for text boxes (`characters`, `distance`); the default is to retain the existing setting

Parameters

`NUMBER` = *scalars*	Numbers associated with the pens
`COLOUR` = *texts* or *scalars*	Colour to use with each pen unless otherwise specified by the `CSYMBOL`, `CLINE`, `CFILL` or `CAREA` parameters
`LINESTYLE` = *texts* or *scalars*	Style for line used by each pen when joining points
`METHOD` = *string tokens*	Method for determining line (`point`, `line`, `monotonic`, `closed`, `open`, `fill`, `spline`, `polygon`)
`SYMBOL` = *texts*, *scalars*, *pointers* or *matrices*	Defines the plotting symbol for each pen, by a text or scalar for a pre-defined symbol, a pointer for a user-defined symbol, or a matrix to supply a bitmap
`LABELS` = *texts* or *factors*	Define labels that will be printed alongside the plotting symbols
`ROTATION` = *scalars* or *variates*	Rotation required for the plotting symbols and labels (in degrees)
`JOIN` = *string tokens*	Order in which points are to be joined by each pen (`ascending, given`)
`BRUSH` = *scalars*	Number of the type of area filling used with each pen when drawing pie charts or histograms (unavailable in Genstat *for Windows*)
`FONT` = *texts* or *scalars*	Font to be used for any text written by each pen
`THICKNESS` = *scalars*	Thickness with which any lines are drawn by each pen
`SIZEMULTIPLIER` = *scalars* or *variates*	Multiplier used in the calculation of the size in which to draw symbols and labels by each pen unless otherwise specified by `SMSYMBOL` or `SMLABEL`
`CSYMBOL` = *texts* or *scalars*	Colour to use with each pen when drawing symbols

CLINE = *texts* or *scalars*	Colour to use with each pen when drawing lines
CFILL = *texts* or *scalars*	Colour to use with each pen when filling areas inside hollow symbols
CAREA = *texts* or *scalars*	Colour to use with each pen when filling areas inside polygons and bars of histograms
SMSYMBOL = *scalars* or *variates*	Multiplier used in the calculation of the size in which to draw symbols by each pen
SMLABEL = *scalars* or *variates*	Multiplier used in the calculation of the size in which to draw labels by each pen
DFSPLINE = *scalars*	Number of degrees of freedom to use when METHOD=spline
YMISSING = *string token*	How to treat missing y-values when METHOD=spline (break, interpolate)
XMISSING = *string token*	How to treat missing x-values when METHOD=spline (break, ignore)
YLPOSITION = *string token*	How to position labels in the y-direction with respect to the points (above, centre, below, automatic)
XLPOSITION = *string token*	How to position labels in the x-direction with respect to the points (left, centre, right, automatic)
YLSIZE = *scalars* or *variates*	Sizes of the y-direction of the text boxes into which to plot labels
XLSIZE = *scalars* or *variates*	Sizes of the x-direction of the text boxes
YLOFFSET = *scalars* or *variates*	Offsets in the y-direction of the text boxes
XLOFFSET = *scalars* or *variates*	Offsets in the x-direction of the text boxes
BARTHICKNESS = *scalars*	Thickness with which any error bars are drawn by each pen
BARCAPWIDTH = *scalars*	Width of the cap drawn by each pen at the top and bottom of any error bars
DESCRIPTION = *texts*	Description for points plotted by the pen, to be used by the Data Information tool in the Graphics Viewer
TSYMBOL = *scalars*	Defines the transparency of symbols drawn by each pen, on a scale of 0 (opaque) to 255 (completely transparent)
TLINE = *scalars*	Defines the transparency of lines drawn by each pen
TFILL = *scalars*	Defines the transparency to use when filling areas inside hollow symbols with each pen
TAREA = *scalars*	Defines the transparency to use when filling areas inside polygons and bars of histograms with each pen
SAVE = *pointers*	Saves details of the current settings for the pen concerned

Graphical displays are drawn using graphical *pens*. Certain pens are used by default, or you can specify other pens, as described in the preceding sections. The attributes of each pen, such as colour and symbol-type, determine how they are used to generate output. The initial defaults for each pen are device-specific, and are described at the end of this subsection. The PEN directive can be used to change these attributes so that you can modify the resulting display. Different attributes are relevant for different types of output, for example symbols and labels are used only within DGRAPH and D3GRAPH (and the graphics procedures that use them to construct their plots).

The NUMBER parameter lists the numbers of the pens, in the range 1 to 256 or −1 to −12, that you wish to redefine. By default, any aspects of these pens that are not set explicitly retain the values that they had immediately before the PEN statement. Alternatively, you can specify option RESET=yes to reset their definitions to the default values defined by Genstat at the start of each job.

Pens 1 to 256 are used for the information that is plotted in a graph (points, lines, and so on). In most of the graphics commands, the default is to use these pens in succession for the different structures that are plotted, so that the various data sets can easily be distinguished. The negatively numbered pens are used as the initial defaults for the axes and their associated marks and labels (see `XAXIS`; 6.9.4), and for gridlines, the overall title and the key (see `FRAME`; 6.9.3), or for default gridlines in shade plots (see `DSHADE`; 6.4.2), or for default outlines in histograms (see `DHISTOGRAM`; 6.3.1), bar charts (see `BARCHART`; 6.3.2) and pie charts (see `DPIE`; 6.6.1), or for error bars (see `BARCHART`; 6.3.2), or for the overall title (see `DSTART`; 6.8.2). They cannot be used for any other purposes.

The `COLOUR`, `CSYMBOL`, `CLINE`, `CFILL` and `CAREA` parameters specify the colours to be used by the pen. The `COLOUR` parameter can be used to define the colour for anything plotted by the pen, while the other parameters define specific aspects (overriding any setting of `COLOUR`): `CSYMBOL` defines the colour to be used for drawing symbols, `CLINE` defines the colour for lines, `CFILL` defines the colour for filling areas, and `CAREA` defines the colour for filling areas inside polygons and bars of histograms. The parameters can be set any of the following: a text containing the name of one of Genstat's pre-defined colours; a scalar containing a number defining a colour using the RGB system; or a hexadecimal digit defined in a string of the form `'#rgb'`, `'0xrgb'` or `'0Xrgb'` where `rgb` are the pairs of hexadecimal digits 00-FF that give the red, green and blue intensities of the colour respectively. For example, `'#FF0000'`, `'#00FF00'` and `'#0000FF'` give pure red, green and blue respectively. The leading zeros can be dropped, so `'#FF00'` and `'#FF'` also define green and blue respectively. You can use the `RGB` function (4.2.12) to construct these colour numbers from their red, green and blue components: for example

```
    CALCULATE xgold = RGB(255; 215; 0)
    PEN 2; CSYMBOL=xgold
```

sets `xgold` to the colour gold (which has red, green and blue values 255, 215 and 0 respectively) and uses this as the colour for symbols drawn in future by pen 2. The numbers give you access to the complete spectrum supported by most colour graphics devices. (Note, though, that they will automatically be mapped onto a grey scale if the device is defined with a grey-scale palette; see `DEVICE`). Alternatively, the pre-defined colours define the standard colours used by many web browsers, and mainly use the same names. The names, and their corresponding red, green and blue values, are listed in 6.9.9. They can be given in either upper or lower case, or in any mixture, but they must not be abbreviated.

There are two special strings that can be used for colours. The string `'background'` uses the colour defined in the `BACKGROUND` option or parameter of `FRAME`. The string `'match'` which can be used with `CFILL` to take the colour from `CSYMBOL`, or with `CAREA` to take the colour from `CLINE`. For example,

```
    PEN 1,2,3; COLOUR='red','blue','green'; CFILL='match'
    PEN 4,5,6; CLINE='red','blue','green'; CAREA='match'
```

plots filled symbols in the same colour as their outlines for pens 1 to 3, and filled areas in the same colour as their outlines for pens 4 to 5. Note, `COLOUR` sets all of `CSYMBOL`, `CLINE` and `CAREA` to the same value, so you only need to set `CFILL='match'` to set all colours of a pen to the same value. Also, if you want all your symbols filled, you could specify

```
    PEN 1...256; CFILL='match'
```

You can also use the number -1 to specify the background colour. A missing value represents a hollow symbol for `CFILL` or the background colour for `CSYMBOL`, `CLINE` and `CAREA`.

The `TSYMBOL`, `TLINE`, `TFILL` and `TAREA` parameters accompany the parameters `CSYMBOL`, `CLINE`, `CFILL` and `CAREA`, respectively, and define the transparency of the corresponding colours. Their values are on a scale of 0 (opaque) to 255 (completely transparent). The pens have initial defaults of 0.

The `SYMBOL` parameter defines the symbol that is drawn at each point, for example by `DGRAPH`. You can mark different points with different symbols (for example to indicate groupings in the data) by setting the `PEN` parameter of `DGRAPH` to a variate or factor specifying a pen with the appropriate symbol for each point.

Genstat provides a choice of standard symbols that can be specified either by giving the name (in a text with a single value), or the number (in a scalar). See Figure 6.9.8a and the list below.

1	'Cross'
2	'Circle'
3	'Plus'
4	'Star'
5	'Square'
6	'Diamond'
7	'Triangle'
8	'Nabla'
9	'Asterisk'
10	'Minus'
11	'Heavyminus'
12	'Heavyplus'
13	'Heavycross'
14	'Smallcircle'
15	'Tinycircle'
16	'Female'
17	'Male'
18	'Rhombus'
19	'Circlecross'
20	'Circleplus'
21	'Squarecross'
22	'Squareplus'
-1	'Sphere'
-2	'Cone'
-3	'Cylinder'
-4	'Cube'

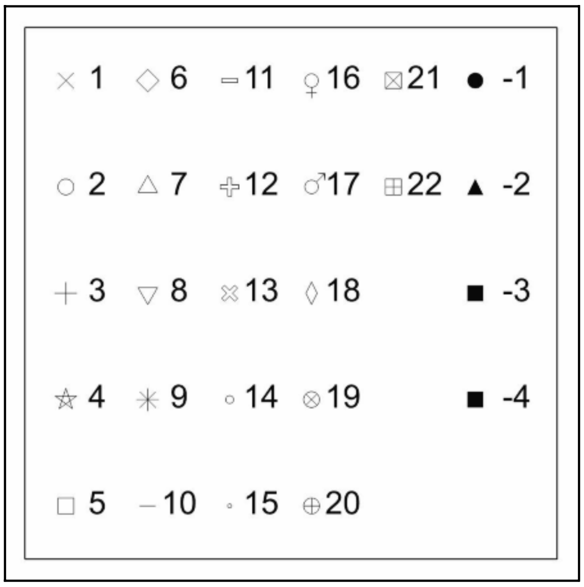

Figure 6.9.8a

The final four symbols (numbered -1 to -4) are intended mainly for 3-dimensional plots, and may not be available on some devices. You can set `SYMBOL='none'` or `SYMBOL=0` if you do not want to plot symbols at the data points, as for example if you only want to draw a line through the points. You can also use `SYMBOL=0` together with the `LABELS` parameter (described below) to plot a character at the data points instead of a symbol. For example

```
PEN 1; SYMBOL='none'; LABEL='A'
```

or

```
PEN 1; SYMBOL=0; LABEL='A'
```

will identify the points plotted by pen 1 with the letter `A`.

To define a symbol of your own, you can set `SYMBOL` to a pointer containing a pair of variates defining the coordinates of a set of points to be joined by straight line segments. The points should be within a notional square with bounds -1.0 to 1.0 in each direction. The square is centred on the data point, and scaled to the same size as the standard symbols. Missing values can be included in the coordinates, to use separate pen strokes to draw the line segments. The final possibility is to set `SYMBOL` to a matrix of RGB colour values, representing a bitmap.

User-defined symbols are illustrated in Example 6.9.8a.

Example 6.9.8a

```
FRAME    [GRID=xy] 1...4; YLOWER=0.75; YUPPER=1.0; \
         XLOWER=0.0,0.25,0.5,0.75; XUPPER=0.25,0.5,0.75,1.0; \
         YMLOWER=0.05; YMUPPER=0.01; XMLOWER=0.05; XMUPPER=0.01;\
         PENGRID=29
PEN      29; LINESTYLE=7; COLOUR='black'
XAXIS    1,2,3,4; LOWER=-1.2,0.8; UPPER=1.2,3.2; MARKS=1
YAXIS    1,2,3,4; LOWER=-1.2,0.8; UPPER=1.2,3.2; MARKS=1
VARIATE Diamond[1]; VALUES=!(-1,0,1,0,-1)
&        Diamond[2]; VALUES=!(0,-0.5,0,0.5,0)
PEN      1; COLOUR='black'; METHOD=line; SYMBOL=0; JOIN=given; THICKNESS=2
&        2; COLOUR='black'; SYMBOL=Diamond; SMSYMBOL=2
DGRAPH   [WINDOW=1; KEYWINDOW=0] Diamond[1]; Diamond[2]; PEN=1
&        [WINDOW=2; SCREEN=keep] 1,2,3; 1,3,2; PEN=2
VARIATE Arrow[1]; VALUES=!(0.0,1.0,0.75,*,1.0,0.75)
&        Arrow[2]; VALUES=!(0.0,0.0,-0.25,*,0.0,0.25)
PEN      3; COLOUR='black'; SYMBOL=Arrow; SIZE=!(2,2.5,3,2);\
         ROTATION=!(0,45,90,180); SMSYMBOL=3
DGRAPH   [WINDOW=3; KEYWINDOW=0; SCREEN=keep] Arrow[1]; Arrow[2]; PEN=1
&        [WINDOW=4] !(1.0,2.7,2.0,1.6); !(1.4,1.8,2.2,2.6); PEN=3
```

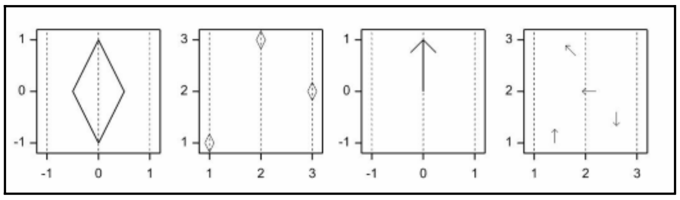

Figure 6.9.8b

The definition of the arrow symbol in lines 15 and 16 illustrates how missing values can be included so that separate pen strokes are used draw line segments. The plot produced by this example is shown in Figure 6.9.8b.

You can mark different points with different symbols (for example to indicate groupings in the data) by setting the PEN parameter of DGRAPH (6.2.1) to a variate or factor specifying a pen with the appropriate symbol for each point.

You can also use the LABELS parameter to label each point with a string or a number. The LABELS parameter can be set to a single string to plot the same label at every point, or text structure with same number of values as the Y and X variates that are being plotted. Alternatively LABELS can be set to a factor; the factor labels are then used, if available, otherwise the levels. This provides another means of representing grouped data. The positioning of the labels with respect to the points is controlled by the YLPOSITION and XLPOSITION parameters. The initial default is to determine the positions automatically according to their type (e.g. labels for points, or for tick marks on the y-axis, or on the x-axis, and so on).

The graphical symbols are drawn so that they are centred at the specified position. If LABELS are specified they are aligned alongside the markers, unless you have set SYMBOLS=0 to suppress the markers, in which case the labels start from the specified (x,y) position. For compatibility with previous releases of Genstat you can also set SYMBOLS to a factor or text, which has the same effect as setting LABELS with SYMBOLS=0.

The Genstat Graphics Viewer with Genstat *for Windows* has a "Data Information" tool that allows you to display information about each point when you place the cursor over the point. If you want to replace the default information, you can set the DESCRIPTION parameter to a text (with one line for each point) containing your own information.

The METHOD parameter specifies the type of object to be plotted: points, lines or filled polygons. The initial default for every pen, METHOD=point, will result in points being plotted using the corresponding symbols, labels, colours and fonts. Various types of line can be drawn through the plotted points; either straight lines (line and polygon) or smooth curves (monotonic, open, closed and spline). The line and polygon settings differ in that, with polygon, a line is drawn also to connect the first and last points. The monotonic setting specifies that a smooth single-valued curve is to be drawn through the data points. The name is derived from the requirement that the x-values (rather than the fitted curve) must be strictly monotonic, so that there is only one y-value for each distinct x-value. To ensure this, a copy of the data is made and sorted before the curve is fitted. This setting is recommended for plotting curves fitted to data, for example with FITCURVE. You should ensure that the points are close enough for the plotted line to be a reasonable approximation. When you know the functional form of the curve, it may be advantageous to calculate extra points. The open and closed settings specify that a smooth, possibly multi-valued, curve is to be drawn through the data points, using the method of McConalogue (1970); the resulting curve is rotationally invariant, although it is not invariant under scaling. The closed setting connects the last point to the first. McConalogue's method (open or closed) is more suited to the situation where the plotted curve is intended to represent the shape of an object. Alternatively, the spline setting plots a smoothing spline fitted through the points. The DFSPLINE parameter specifies how many degrees of freedom to use in the spline (initial default 4). The YMISSING parameter controls whether to break the spline at a missing y-value or to interpolate y-value, and the XMISSING parameter controls whether to break the spline at a missing x-value or to ignore the point; the initial default for both parameters is to break the spline. The setting METHOD=fill joins the data points by straight lines to produce one or more polygons. Each polygon is then filled in the colour specified by CFILL (see below). The plotting method also determines how contours will be drawn, as described in 6.4.1. Also, the combination of SYMBOLS=0 and METHOD=point will produce no plotting at all (and no warning) within DGRAPH.

If the requested plotting method produces a line through the points, the LINESTYLE parameter will specify what sort of line is drawn (for example a solid, dotted or dashed line). The type of style can be specified either by giving the name (in a text with a single value), or the number (in a scalar).

1	'Solid'
2	'Dot' or '.'
3	'Dash' or '-'
4	'Dotdash' or '.-'
5	'Tightdash' or 'T-'
6	'Longdash' or 'L-'
7	'Shortdash' or 'S-'
8	'Closedot' or 'C-'
9	'Finedot' or 'F.'
10	'Doubledotdash' or '..-'

Each text can all be abbreviated to the minimum number of characters required to distinguish it from the earlier styles. Figure 6.9.8c illustrates the line styles available.

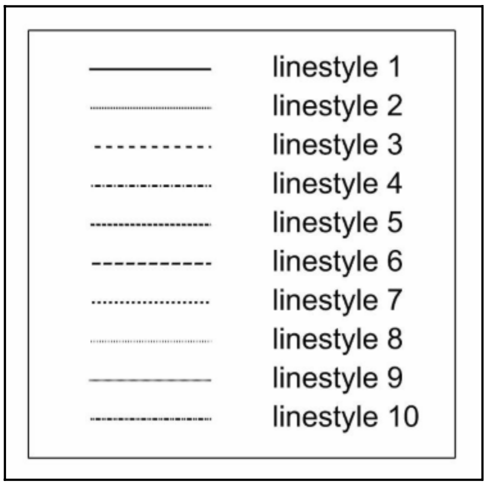

Figure 6.9.8c

The JOIN parameter controls the order in which points are connected when lines are to be drawn or the points define a polygon to be shaded. Given requests that the data are to be plotted in the order in which they are stored, whereas ascending implies that the data are copied and sorted so that the x-values are in ascending order before plotting. This parameter is ignored when METHOD=monotonic, as this requires that the data must always be sorted.

The BRUSH parameter was used on some monochrome devices to controls how areas are shaded when METHOD is set to fill, or when plotting histograms and pie charts. There were 16 available patterns indicated by the integers 1 to 16, as shown in Figure 6.9.8d. In general, the higher the number, the denser the hatching. In Genstat *for Windows* BRUSH is unavailable, and the areas are shaded in full. The CFILL parameter defines which colour is used by the pen to fill the areas.

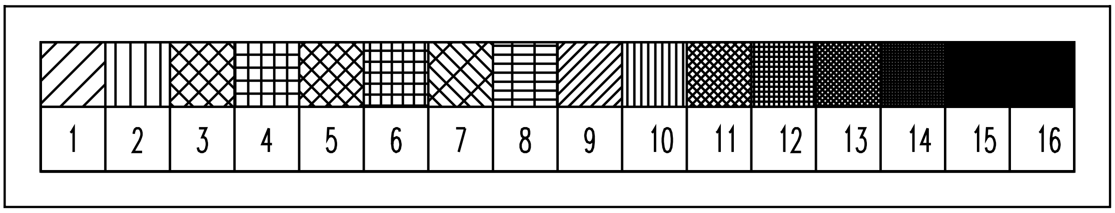

Figure 6.9.8d

The `THICKNESS` parameter allows you to specify an amount by which the standard thickness of plotted lines is to be multiplied. This allows you to increase the thickness of lines, perhaps to highlight some feature of a plot, as illustrated in the contour plot in Figure 6.4.1b. You can also use thickness to emphasize the axes, by redefining the appropriate pen. For some devices, it is not possible to control the thickness of plotted lines; the `THICKNESS` parameter is then ignored.

The default size of characters and symbols is determined from the dimensions of the current window. The `SIZEMULTIPLIER` parameter can be used to modify the sizes of both of these, by specifying a value by which this default size is to be multiplied. Alternatively, you can use the `SMSYMBOL` parameter to modify just the symbol size, or the `SMLABEL` parameter to modify just the size of characters in labels. For example when plotting a graph in a small window you may wish to increase the size of annotation in order to make it legible. They can each be set to a scalar, or to a variate to allow the different points to be scaled in different ways.

The `ROTATION` parameter controls the angle (in degrees) at which to plot text or user-defined symbols. The initial setting of zero will produce text "conventionally" orientated. You can set `ROTATION` to a scalar value that will apply to all points, or to a variate that allows a different angle to be used at each point. `ROTATION` is used in line 18 of Example 6.9.8a to plot a user-defined symbol at different angles.

The `FONT` parameter defines the font family to be used by each pen to plot textual information, for example, in titles, axis annotation, plotting symbols and keys. This can be set to a text containing the name of a font family, or to a scalar containing an integer between 1 and 25. The initial default for each pen is to use the *default graphics font*, which can be defined either by using menus in the Genstat Client or Graphics Viewer, or by using the `DFONT` directive (6.9.12). You can find out the names of the fonts, available to specify in a text, by looking at any of the controls for specifying fonts in the Client or Graphics Viewer. The integers refer to fonts that should always be available. You can list these using the `DHELP` procedure (6.9). Font 1 has a special status, in that it automatically maps to the currently-defined default graphics font. If you change the default graphics font, this will be used as the default font in any graphs that are then displayed or redisplayed, including those that have been stored in Genstat graphics meta files (i.e. files with the `gmf` suffix). If you specify a font that is unavailable on your computer, the default font is used instead.

The current settings of each pen can be saved in a pointer supplied by the `SAVE` parameter. The elements of the pointer are labelled to identify the components, as shown in Example 6.9.8b.

Example 6.9.8b

```
   2  PEN   8; LABELS='observation'; SYMBOL=8; JOIN=given; SAVE=Pen8
   3  PRINT [RLWIDTH=19; ORIENTATION=across] Pen8[]; FIELDWIDTH=18

    Pen8['boxunits']        characters
      Pen8['colour']          16747520
   Pen8['linestyle']                 *
      Pen8['method']             point
     Pen8['symbol']                  8
      Pen8['labels']       observation
    Pen8['rotation']            0.00
        Pen8['join']           given
       Pen8['brush']               *
        Pen8['font']               1
   Pen8['thickness']            1.00
        Pen8['size']            1.00
     Pen8['csymbol']        16747520
       Pen8['cline']        16747520
       Pen8['cfill']               *
       Pen8['carea']        16747520
    Pen8['smsymbol']            1.00
     Pen8['smlabel']            1.00
    Pen8['dfspline']            4.00
    Pen8['ymissing']           break
    Pen8['xmissing']           break
  Pen8['ylposition']       automatic
  Pen8['xlposition']       automatic
Pen8['barthickness']            1.00
 Pen8['barcapwidth']            1.00
     Pen8['tsymbol']            0.00
       Pen8['tline']            0.00
       Pen8['tfill']            0.00
       Pen8['tarea']            0.00
    Pen8['fontname']           Arial
```

Note that the saved values for line style and brush style are missing values. This is how the initial default settings are represented; the actual values used for these attributes when plotting will depend on the output device, unless they are set explicitly (as with `SYMBOL` in this example).

6.9.9 Colours

The names of the standard pre-defined Genstat colours are listed below with their corresponding red, green and blue values for use e.g. in the `RGB` function (4.2.12).

Red colors

IndianRed	RGB(205; 92; 92)
LightCoral	RGB(240; 128; 128)
Salmon	RGB(250; 128; 114)
DarkSalmon	RGB(233; 150; 122)
LightSalmon	RGB(255; 160; 122)
Crimson	RGB(220; 20; 60)
Red	RGB(255; 0; 0)
FireBrick	RGB(178; 34; 34)
DarkRed	RGB(139; 0; 0)

Pink colors

Pink	RGB(255; 192; 203)
LightPink	RGB(255; 182; 193)
HotPink	RGB(255; 105; 180)
DeepPink	RGB(255; 20; 147)
MediumVioletRed	RGB(199; 21; 133)
PaleVioletRed	RGB(219; 112; 147)

Orange colors

LightSalmon	RGB(255; 160; 122)
Coral	RGB(255; 127; 80)
Tomato	RGB(255; 99; 71)
OrangeRed	RGB(255; 69; 0)
DarkOrange	RGB(255; 140; 0)
Orange	RGB(255; 165; 0)

Yellow colors

Gold	RGB(255; 215; 0)
Yellow	RGB(255; 255; 0)
LightYellow	RGB(255; 255; 224)
LemonChiffon	RGB(255; 250; 205)

LightGoldenrodYellow	RGB(250; 250; 210)
PapayaWhip	RGB(255; 239; 213)
Moccasin	RGB(255; 228; 181)
PeachPuff	RGB(255; 218; 185)
PaleGoldenrod	RGB(238; 232; 170)
Khaki	RGB(240; 230; 140)
DarkKhaki	RGB(189; 183; 107)

Purple colors

Lavender	RGB(230; 230; 250)
Thistle	RGB(216; 191; 216)
Plum	RGB(221; 160; 221)
Violet	RGB(238; 130; 238)
Orchid	RGB(218; 112; 214)
Fuchsia	RGB(255; 0; 255)
Magenta	RGB(255; 0; 255)
MediumOrchid	RGB(186; 85; 211)
MediumPurple	RGB(147; 112; 219)
BlueViolet	RGB(138; 43; 226)
DarkViolet	RGB(148; 0; 211)
DarkOrchid	RGB(153; 50; 204)
DarkMagenta	RGB(139; 0; 139)
Purple	RGB(128; 0; 128)
Indigo	RGB(75; 0; 130)
SlateBlue	RGB(106; 90; 205)
DarkSlateBlue	RGB(72; 61; 139)

Green colors

GreenYellow	RGB(173; 255; 47)
Chartreuse	RGB(127; 255; 0)
LawnGreen	RGB(124; 252; 0)
Lime	RGB(0; 255; 0)
LimeGreen	RGB(50; 205; 50)
PaleGreen	RGB(152; 251; 152)
LightGreen	RGB(144; 238; 144)
MediumSpringGreen	RGB(0; 250; 154)
SpringGreen	RGB(0; 255; 127)
MediumSeaGreen	RGB(60; 179; 113)
SeaGreen	RGB(46; 139; 87)
ForestGreen	RGB(34; 139; 34)
Green	RGB(0; 128; 0)
DarkGreen	RGB(0; 100; 0)
YellowGreen	RGB(154; 205; 50)
OliveDrab	RGB(107; 142; 35)
Olive	RGB(128; 128; 0)
DarkOliveGreen	RGB(85; 107; 47)
MediumAquamarine	RGB(102; 205; 170)
DarkSeaGreen	RGB(143; 188; 143)
LightSeaGreen	RGB(32; 178; 170)
DarkCyan	RGB(0; 139; 139)
Teal	RGB(0; 128; 128)

Blue colors

Aqua	RGB(0; 255; 255)
Cyan	RGB(0; 255; 255)
LightCyan	RGB(224; 255; 255)
PaleTurquoise	RGB(175; 238; 238)
Aquamarine	RGB(127; 255; 212)
Turquoise	RGB(64; 224; 208)
MediumTurquoise	RGB(72; 209; 204)
DarkTurquoise	RGB(0; 206; 209)
CadetBlue	RGB(95; 158; 160)
SteelBlue	RGB(70; 130; 180)
LightSteelBlue	RGB(176; 196; 222)

PowderBlue	RGB(176; 224; 230)
LightBlue	RGB(173; 216; 230)
SkyBlue	RGB(135; 206; 235)
LightSkyBlue	RGB(135; 206; 250)
DeepSkyBlue	RGB(0; 191; 255)
DodgerBlue	RGB(30; 144; 255)
CornflowerBlue	RGB(100; 149; 237)
MediumSlateBlue	RGB(123; 104; 238)
RoyalBlue	RGB(65; 105; 225)
Blue	RGB(0; 0; 255)
MediumBlue	RGB(0; 0; 205)
DarkBlue	RGB(0; 0; 139)
Navy	RGB(0; 0; 128)
MidnightBlue	RGB(25; 25; 112)

Brown colors

Cornsilk	RGB(255; 248; 220)
BlanchedAlmond	RGB(255; 235; 205)
Bisque	RGB(255; 228; 196)
NavajoWhite	RGB(255; 222; 173)
Wheat	RGB(245; 222; 179)
BurlyWood	RGB(222; 184; 135)
Tan	RGB(210; 180; 140)
RosyBrown	RGB(188; 143; 143)
SandyBrown	RGB(244; 164; 96)
Goldenrod	RGB(218; 165; 32)
DarkGoldenrod	RGB(184; 134; 11)
Peru	RGB(205; 133; 63)
Chocolate	RGB(210; 105; 30)
SaddleBrown	RGB(139; 69; 19)
Sienna	RGB(160; 82; 45)
Brown	RGB(165; 42; 42)
Maroon	RGB(128; 0; 0)

White colors

White	RGB(255; 255; 255)
Snow	RGB(255; 250; 250)
Honeydew	RGB(240; 255; 240)
MintCream	RGB(245; 255; 250)
Azure	RGB(240; 255; 255)
AliceBlue	RGB(240; 248; 255)
GhostWhite	RGB(248; 248; 255)
WhiteSmoke	RGB(245; 245; 245)
Seashell	RGB(255; 245; 238)
Beige	RGB(245; 245; 220)
OldLace	RGB(253; 245; 230)
FloralWhite	RGB(255; 250; 240)
Ivory	RGB(255; 255; 240)
AntiqueWhite	RGB(250; 235; 215)
Linen	RGB(250; 240; 230)
LavenderBlush	RGB(255; 240; 245)
MistyRose	RGB(255; 228; 225)

Grey colors

Gainsboro	RGB(220; 220; 220)
LightGray or LightGrey	RGB(211; 211; 211)
Silver	RGB(192; 192; 192)
DarkGray or DarkGrey	RGB(169; 169; 169)
Gray or Grey	RGB(128; 128; 128)
DimGray or DimGrey	RGB(105; 105; 105)
LightSlateGray or LightSlateGrey	RGB(119; 136; 153)
SlateGray or SlateGrey	RGB(112; 128; 144)

```
DarkSlateGray or DarkSlateGrey
                          RGB( 47;  79;  79)
Black                     RGB(  0;   0;   0)
```

In addition the string `'Background'` can be used to refer to the background colour that has been defined (e.g. by `FRAME`) for the particular part of the screen where the pen is being used. Another useful setting is the string `'Transparent'`. This is used as the initial default for the background colours of the graphics windows.

The initial defaults for the colours of the pens (6.9.8) defines a standard sequence of colours, that is also used to set default colours in procedures like `AGRAPH`. This can be accessed using the `GETRGB` procedure.

GETRGB procedure

Gets the RGB values and names of the initial default graphics colours of the Genstat pens (R.W. Payne).

No options

Parameters

`COLOUR` = *scalars* or *variates*	Colour numbers
`RGB` = *scalars* or *variates*	RGB values
`NAME` = *texts*	Names of nearest colours

The `COLOUR` parameter specifies a scalar or variate containing the pen number(s) whose initial default colours are required. The `RGB` parameter saves a scalar or variate containing the corresponding colours, expressed as RGB values (see `PEN`). The `NAME` parameter saves a text containing the name of the nearest colour.

Example 6.9.9a uses `GETRGB` to display the first 32 colours in the sequence.

Example 6.9.9a

```
   2  VARIATE [VALUES=1...32] Ncolour
   3  GETRGB  Ncolour; RGB=RGB; NAME=Name
   4  PRINT   Ncolour,Name,RGB; DECIMALS=0

  Ncolour           Name        RGB
        1          Black          0
        2            Red   16711680
        3      LimeGreen    3329330
        4           Blue        255
        5           Aqua      65535
        6        Fuchsia   16711935
        7         Yellow   16776960
        8     DarkOrange   16747520
        9     Chartreuse    8388352
       10    SpringGreen      65407
       11     DodgerBlue    2003199
       12     BlueViolet    9055202
       13       DeepPink   16716947
       14        DimGray    6908265
       15       DarkGray   11119017
       16      IndianRed   13458524
       17    YellowGreen   10145074
       18       SeaGreen    3050327
       19      Turquoise    4251856
       20  DarkSlateBlue    4734347
       21   MediumOrchid   12211667
       22        DarkRed    9109504
       23          Green      32768
       24           Navy        128
       25         Salmon   16416882
```

```
 26        PaleGreen    10025880
 27 MediumSlateBlue     8087790
 28   DarkSlateGray     3100495
 29         Thistle    14204888
 30            Gray     8421504
 31          Silver    12632256
 32          Maroon     8388608
```

Alternatively, you can define your own sequences of colours using the `DCOLOURS` procedure.

`DCOLOURS` procedure

Forms a band of graduated colours for graphics (P.W. Goedhart).

Options

`METHOD` = *string token*	Type of colour band required (`spectral`, `blackbody`, `linear`); default `line`
`PLOT` = *string token*	What to plot (`testgraph`); default `*`

Parameters

`START` = *scalar* or *text*	Start value for the colour band; default `*` gives an appropriate default for the `METHOD` concerned
`END` = *scalar*, *text* or *variate*	End value(s) for the colour band; default `*` gives an appropriate default for the `METHOD` concerned
`GAMMA` = *scalar* or *variate*	The gamma-correction exponent(s) for the colour band; default 1
`NCOLOURS` = *scalar* or *variate*	Number(s) of colours in the colour band; default 20
`RGB` = *variates*	Saves the RGB colour values of each colour band
`RED` = *variates*	Saves the red component of the RGB colour values
`GREEN` = *variates*	Saves the green component of the RGB colour values
`BLUE` = *variates*	Saves the blue component of the RGB colour values
`TITLE` = *text*	General title for each test graph; default forms an informative title automatically
`WINDOW` = *scalar*	Window number for each test graph; default 1
`SCREEN` = *string token*	Whether to clear the screen before plotting each test graph or to continue plotting on the old screen (`clear`, `keep`); default `clea`

Procedure `DCOLOURS` creates a colour band by interpolating between start and end colour values. You can save the RGB colours of the band, in a variate, using the `RGB` parameter. Alternatively, you can save the red, green and blue components of the colours using the `RED`, `GREEN` and `BLUE` parameters (again in variates).

A test graph displaying the colour band can be requested by setting option `PLOT=testgraph`. The `WINDOW` parameter supplies the window number for the plot (default 1). The `TITLE` parameter can supply a title for the test graph; if this is not set, a suitable title is generated automatically. You can set parameter `SCREEN=keep` to plot the test graph on an existing screen; by default the screen is cleared first.

The `METHOD` option provides a choice of three different types of colour band. The default, `METHOD=linear`, forms the colours by interpolating between start and end RGB values. The start value is specified by the `START` parameter, as either a scalar defining an RGB colour value, or a text containing the name of one of the pre-defined Genstat colours (see the `PEN` directive for the available names, or search for "Graphics Colours" in the on-line help). You can set the `END` parameter to a single scalar or text (giving either the RGB value or the name of the colour) to define the band as a single sequence of colours. Alternatively, you can define a variate or a text with several values to form the band from several sequences of colours. At each END colour, DCOLOURS then begins a new sequence running from that colour to the next END colour. The default values for START and END are 'white' and 'black'.

Setting METHOD=spectral forms an approximate rainbow spectrum for wavelengths between 380 nm and 780 nm. There can now be only a single sequence of colours. The START and END parameters specify the start and end wavelengths, as scalars, with default values of 380 and 780.

The final setting, METHOD=blackbody, forms colours of hot objects with temperatures between 500 K and 11000 K. Again, only a single sequence of colours is allowed. The START and END parameters specify the start and end temperatures, as scalars, with default values of 500 and 11000.

The NCOLOURS parameter specifies the number of colours in each sequence of colours, as a scalar for the spectral or blackbody methods, or as either a scalar or a variate for the linear method; the default is 20.

The red, green and blue values in each sequence are assumed by default to vary linearly with wavelength, temperature or red/green/blue components. Alternatively, you can use the GAMMA parameter to specify the power for a power transformation (default 1). It must be set to a scalar for the spectral or blackbody methods, and to either a scalar or a variate for the linear method. Its values must lie in the interval [0.25, 4].

The number of values specified by each set of END, GAMMA and NCOLOURS parameters can be different. However, the number of values in the setting of the END parameter determines the number of colour sequences in the band, and the values in the GAMMA setting and NCOLOURS setting are recycled as required.

Example 6.9.9b uses DCOLOURS to form a sequence of colours running from red to white and then to blue. The colour map (plotted in window 3 by line 6) is shown in Figure 6.9.9.

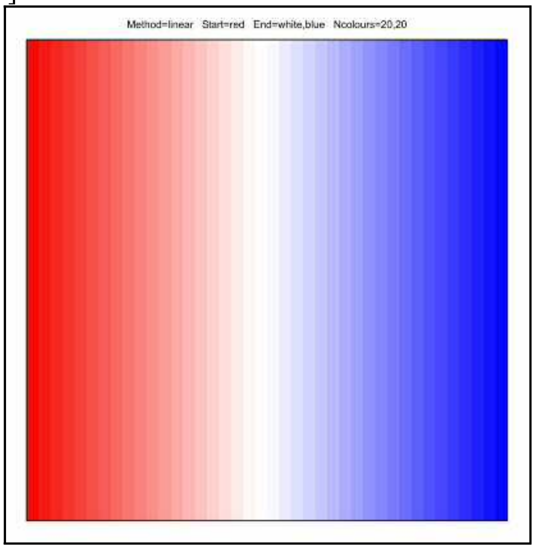

Figure 6.9.9

Example 6.9.9b

```
 5  DCOLOURS [METHOD=linear; PLOT=testgraph] START='red'; END=!t(white,blue);\
 6           WINDOW=3; RGB=RGBseq
 7  PRINT    RGBseq

      RGBseq
    16711680
    16715021
    16718362
    16721960
    16725301
    16728899
    16732240
    16735581
    16739179
    16742520
    16746118
```

```
16749459
16753057
16756398
16759739
16763337
16766678
16770276
16773617
16777215
15921919
15066623
14211327
13421823
12566527
11711231
10855935
10066431
 9211135
 8355839
 7500543
 6711039
 5855743
 5000447
 4145151
 3355647
 2500351
 1645055
  789759
     255
```

6.9.10 Accessing details of the graphics environment

`DKEEP` directive

Saves information from the last plot on a particular device.

No options

Parameters

`DEVICE` = *scalars*	The devices for which information is required, if the scalar is undefined or contains a missing value, this returns the current device number
`WINDOW` = *scalars*	Window about which the information is required; default `*` gives information about the last window
`XLOWER` = *scalars*	Lower bound for the x-axis in last graph in the specified device and window
`XUPPER` = *scalars*	Upper bound for the x-axis in last graph in the specified device and window
`YLOWER` = *scalars*	Lower bound for the y-axis in last graph in the specified device and window
`YUPPER` = *scalars*	Upper bound for the y-axis in last graph in the specified device and window
`ZLOWER` = *scalars*	Lower bound for the z-axis in last graph in the specified device and window
`ZUPPER` = *scalars*	Upper bound for the z-axis in last graph in the specified device and window
`FILE` = *scalars*	Returns the value 1 or 0 to indicate whether a file is required for this device
`DESCRIPTION` = *texts*	Description of the device
`DREAD` = *scalars*	Returns the value 1 or 0 to indicate whether graphical

	input is possible from this device
`ENDACTION` = *texts*	Returns the current `ENDACTION` setting (`'continue'` or `'pause'`)

`DKEEP` provides information that can be used in general programs and procedures to control the graphical output. For the specified device you can determine whether it generates screen output or uses a file, whether graphical input is possible, a description of the device (as printed by `DHELP`; see the start of Section 6.9), the current `ENDACTION` setting, and details of the axis bounds.

The device for which the information is required is specified by the `DEVICE` parameter. If you specify a scalar containing a missing value, this will be set to the number of the current graphics device. You can then test whether an output file is needed and open one accordingly, as shown in Example 6.9.10a.

Example 6.9.10a

```
DEVICE 7
READ   Y,X,Y2

  Identifier   Minimum      Mean   Maximum    Values   Missing
           Y     46.46     68.11     89.95        20         0
           X    0.9400     4.867     8.877        20         0
          Y2     38.00     62.37     82.65        20         0

SCALAR Device
DKEEP  DEVICE=Device; FILE=File; DESCRIPTION=Name
PRINT  Name,Device,File

                       Name      Device        File
JPEG Files (*.jpg, *.jpeg)           7           1

IF File
  OPEN 'Graph.jpg'; CHANNEL=Device; FILETYPE=graphics
ENDIF
```

When writing a procedure you can find out if axes bounds have been set explicitly, using the `SAVE` parameter of `AXES`. This information may then be used when setting up the axes for other graphs. However, if the bounds were not set, but have been evaluated from the data (or if the axes have subsequently been redefined) the information in the save structure will not be of any use. The actual values used when plotting are recorded internally, for each window of each device, and can be accessed using the `XLOWER`, `XUPPER`, `YLOWER`, `YUPPER`, `ZLOWER` and `ZUPPER`, parameters of `DKEEP`.

Example 6.9.10b

```
DGRAPH [WINDOW=5; KEYWINDOW=7] Y; X
" Now set up window 6 to have the same bounds as window 5,
  so that Y2 is plotted on the same scale as Y."
DKEEP  Device; WINDOW=5; YLOWER=Ymin; YUPPER=Ymax; XLOWER=Xmin; \
       XUPPER=Xmax
PRINT  Ymin,Ymax,Xmin,Xmax

    Ymin        Ymax        Xmin        Xmax
   44.29       92.12      0.5431       9.274

XAXIS  6; LOWER=Xmin; UPPER=Xmax
YAXIS  6; LOWER=Ymin; UPPER=Ymax
DGRAPH [WINDOW=6; KEYWINDOW=8; SCREEN=keep] Y2; X
```

```
  29  CLOSE  Device; FILETYPE=graphics; DELETE=yes
```

6.9.11 Storing and recovering the graphics environment

Once you have defined the graphics environment for a particular type of plot, you may want to save it for use with that type of plot in the future. The `DSTORE` allows you to save the current graphics environment settings in an external file, and the `DLOAD` directive allows you to read them back into Genstat.

`DSAVE` directive

Saves the current graphics environment settings to an external file.

No options

Parameters

`FILENAME` = *text*	File in which to save the environment settings
`DESCRIPTION` = *text*	Description for these settings

`DLOAD` directive

Loads the graphics environment settings from an external file.

No options

Parameter

text	File from which to lead the environment settings

6.9.12 The `DFONT` directive

`DFONT` directive

Defines the default font for high-resolution graphics.

No options

Parameter

text	specifies or saves the default graphics font

`DFONT` allows you to set the default font that is used to plot textual information by the Genstat Graphics Viewer, as an alternative to the use of menus in the Genstat Client or Graphics Viewer.

It has a single, unnamed, parameter, which can be set to text structure containing a single string. If that string is not missing (or null), it specifies the name of the font family to be used as the default. For example,

```
    DFONT 'Calibri'
```

The name can be specified in upper or lower case, or in any mixture. You can find out the available fonts by looking at any of the controls for specifying fonts in the Client or Graphics Viewer.

If the text contains a missing string, it is redefined to contain the name of the font family currently used as the default. It is also defined as a text containing the current default, if you specify either a text with no values or an undeclared data structure, as shown in Example 6.9.12.

Example 6.9.12

```
   2  TEXT  Currentdefault
   3  DFONT Currentdefault
   4  PRINT Currentdefault

Currentdefault
         Arial
```

Finally, if you specify `DFONT` without setting the parameter, it sets the default font back to the standard Genstat font i.e. Arial.

The change takes effect only when information about the new default is received by the Graphics Viewer. Afterwards this will be used in any graphs with the default font that are displayed or redisplayed, including those that have been stored in Genstat graphics meta files (i.e. files with the `gmf` suffix).

6.10 Line-printer graphics

Prior to Release 10.1 there were three directives for line-printer output: `GRAPH`, `HISTOGRAM` and `CONTOUR`. In Release 10.1, these were given a prefix `LP` for clarity, to become `LPGRAPH` (6.10.1), `LPHISTOGRAM` (6.10.2) and `LPCONTOUR` (6.10.3). The original names `GRAPH`, `HISTOGRAM` and `CONTOUR` are currently retained as synonyms, but they may be phased-out or used for high-resolution plots in future releases.

The directives have options and parameters to modify the annotation, the symbols used, the size of plot, and so on. Several options apply generally to all three directives and are described now. Others are more specific and are left until the descriptions of the relevant directives.

Normally, output goes to the current output channel, but you can use the `CHANNEL` option to direct it to another (see 3.3). For example, when you are working interactively, you might want to send a graph to a secondary output file so that you can print it later. Unlike some directives (for example, `PRINT`; 3.2) you cannot save the output in a text structure.

The `TITLE` option lets you set an overall title for the output; graphs and contour plots can also have individual axis titles, specified by the `YTITLE` and `XTITLE` options. You can supply the text settings of these options directly, in a string, or give them as the identifier of a pre-defined text structure. For example:

```
LPGRAPH [XTITLE='Nitrogen Applied (kg/ha)'] Yield; Nitrogen
```

or

```
TEXT Experiment
READ [CHANNEL=2; SERIAL=yes; SETNVALUES=yes]\
  Experiment,Data
LPHISTOGRAM [TITLE=Experiment] Data
```

Genstat prints the y-axis title as a column of characters down the left-hand side of a graph or contour plot. New lines are ignored, so that strings within a text are concatenated. Genstat truncates the title if necessary: the maximum possible number of characters is the number of rows of the frame plus 4. The x-axis title is printed below the graph; the maximum number of characters is the number of columns of the frame plus four: long strings are truncated whereas short strings are centred.

6.10.1 The LPGRAPH directive

LPGRAPH directive

Produces point and line graphs using character (i.e. line-printer) graphics.

Options

CHANNEL = *scalar*	Channel number of output file; default is current output file
TITLE = *text*	General title; default *
YTITLE = *text*	Title for y-axis; default *
XTITLE = *text*	Title for x-axis; default *
YLOWER = *scalar*	Lower bound for y-axis; default *
YUPPER = *scalar*	Upper bound for y-axis; default *
XLOWER = *scalar*	Lower bound for x-axis; default *
XUPPER = *scalar*	Upper bound for x-axis; default *
MULTIPLE = *variate*	Numbers of plots per frame; default * i.e. all plots are on a single frame
JOIN = *string token*	Order in which to join points (ascending, given); default asce
EQUAL = *string tokens*	Whether/how to make bounds equal (no, scale, lower, upper); default no
NROWS = *scalar*	Number of rows in the frame; default * i.e. determined automatically
NCOLUMNS = *scalar*	Number of columns in the frame; default * i.e. determined automatically
YINTEGER = *string token*	Whether y-labels integral (yes, no); default no
XINTEGER = *string token*	Whether x-labels integral (yes, no); default no

Parameters

Y = *identifiers*	Y-coordinates
X = *identifiers*	X-coordinates
METHOD = *string tokens*	Type of each graph (point, line, curve, text); if unspecified, poin is assumed
SYMBOLS = *factors* or *texts*	For factor SYMBOLS, the labels (if defined), or else the levels, define plotting symbols for each unit, whereas a text defines textual information to be placed within the frame for METHOD=text or the symbol to be used for each plot for other METHOD settings; if unspecified, * is used for points, with integers 1-9 to indicate coincident points, ' and . are used for lines and curves
DESCRIPTION = *texts*	Annotation for key

The simplest form of the LPGRAPH directive produces a point plot (or scatterplot as it is sometimes called). It can also be used to plot lines and curves, and text can be added for extra annotation. The data are supplied as y- and x-coordinates in separate parameter lists.

In Example 6.10.1a, the identifiers Y and X are variates of equal length; Genstat uses their values in pairs to give the coordinates of the points to be plotted.

Example 6.10.1a

```
 2  VARIATE [VALUES=-16,-7,9,16,7,-8,-12,-5,0,10,4,-4,-3,3,16] X
 3  & [VALUES=0,-14,-12.5,0,14,0,12,0,-10,-9,5,6,-6,-1.5,16] Y
 4  LPGRAPH Y; X

      -+---------+---------+---------+---------+---------+---------+---
      I                                                               I
 15.0 I                                          *              *     I
      I          *                                                    I
      I                                                               I
      I                                                               I
      I                       *             *                         I
      I                                                               I
  0.0 I   *             *    *                                  *     I
      I                                   *                           I
      I                         *                                     I
      I                                                               I
      I                              *                *               I
      I                                             *                 I
-15.0 I                  *                                            I
      -+---------+---------+---------+---------+---------+---------+---
   -18.0     -12.0      -6.0       0.0       6.0      12.0      18.0

                      Y  v. X using symbol *
```

By default, if you specify several identifiers, Genstat plots them all in the same frame a pair at a time; for example

```
LPGRAPH Y[1...3]; X[1,2]
```

superimposes plots of `Y[1]` against `X[1]`, `Y[2]` against `X[2]`, and `Y[3]` against `X[1]`. The usual rules governing the parallel expansion of lists apply here: the length of the `Y` parameter list determines the number of plots within the frame, and the `X` parameter list is recycled if it is shorter. To generate several frames from one `LPGRAPH` statement you can use the `MULTIPLE` option, described below.

The identifiers supplied by the `Y` and `X` parameters need not be variates, but can be any numerical structures: scalars, variates, factors, tables or matrices. The only constraints are that the pairs of structures must have the same numbers of values, and that tables must not have margins.

There are four types of graph available, controlled by the `METHOD` parameter: `point` (the default), `line`, `curve` and `text`.

A `line` plot is one in which each point is joined to the next by a straight line. Alternatively, using the `curve` method, cubic splines are used to produce a smoothed curve through the data points. This does not represent any model fitted in the statistical sense, but as long as the data points are not too widely spaced (especially where the gradient changes quickly) the plotted curve should be a good representation of the underlying function.

By default, Genstat sorts the data so that the x-values are in ascending order before any line or curve is drawn through the points. However, if you set option `JOIN=given`, the points are joined in the order in which they occur in the data; if there are then any missing values there will be breaks in the line at each missing unit.

Plots produced with `METHOD` set to either `line` or `curve` do not include markings for the data points themselves; you should plot these separately if they are required, as shown in Example 6.10.1b. Here `W` is plotted against `V` twice, first with the `curve` method and then with the `point` method. It is best to plot the line first, so that the symbols for individual points will overwrite those used for the line or curve.

Example 6.10.1b

```
VARIATE [VALUES=-0.1,0.1...0.9] V
& [VALUES=5.5,9.9,8.7,2.3,1.3,5.5] W
LPGRAPH [TITLE='Point and curve plot'; NROWS=16; NCOLUMNS=61] W,W; V;\
  METHOD=curve,point; SYMBOLS=*,'X'; DESCRIPTION='Fitted curve   ...',*

                            Point and curve plot

    -+---------+---------+---------+---------+---------+---------+-
    I              .X    ..                                       I
    I             '        ''.                                    I
    I           .'            X                                   I
8.0 I          '               '                                  I
    I         '                 '                                 I
    I       .'                   '                                I
    I      .                      '                               I
    I     X                        '                       .X     I
    I                               '                     .       I
4.0 I                                .                   '        I
    I                                 .                .'         I
    I                                  '              .           I
    I                                   X..         .'            I
    I                                      '......X'              I
    I                                                             I
0.0 I                                                             I
    -+---------+---------+---------+---------+---------+---------+-
  -0.2       0.0       0.2       0.4       0.6       0.8       1.0

                    Fitted curve   ...
                    W  v. V using symbol X
```

The fourth plotting method is `text`. You can use this to place an item of text within a graph as extra annotation. For example:

```
SCALAR Xt,Yt; VALUE=20,10
TEXT [VALUES='Y=aX+b'] T
LPGRAPH Y,Yt; X,Xt; METHOD=line,text; SYMBOLS=*,T
```

This plots a line, defined by the variates `Y` and `X`, as described above. In addition, the text `T` is printed within the frame starting at the coordinates defined by the scalars `Yt` and `Xt`. As these statements show, the `SYMBOLS` parameter then specifies the text that is to be plotted. The text is truncated as necessary, if positioned too close to the edge of the graph.

With other methods `SYMBOL` defines the plotting symbol to be used to mark either points or lines on the graph. The default symbol for points is the asterisk, and for lines is a combination of dots and single quotes: you can see these in the earlier examples. If several points coincide, Genstat replaces the asterisk by a digit between 2 and 9, representing the number of coincidences, with 9 meaning nine or more. For point plots, the `SYMBOLS` parameter can be set to either a text or a factor. If you specify a text with a single string, the string is used to label every point; otherwise, the text must have one string for each point.

By default, Genstat automatically calculates the extent of the axes from the data to be plotted, in such a way that all the data are contained within the frame. You can set one or more of the bounds for the axes by options `YLOWER`, `YUPPER`, `XLOWER` and `XUPPER`. By setting the upper bound of an axis to a value that is less than the lower bound, you can reverse the usual convention for plotting in which the y-values increase upwards and the x-values increase to the right. Setting the options `YINTEGER` and `XINTEGER` constrains the axis markings to be integral, if possible.

The `EQUAL` option allows you to place constraints on the bounds for the axes. The default setting `no` (meaning no constraint) uses the boundary values as set by the options or calculated from the data. The settings `lower` and `upper` constrain the lower or upper bounds of the two axes to be equal: for example, to plot the line *y*=*x* along with the data, setting `EQUAL=lower` will ensure that it will pass through the bottom left-hand corner of the frame. The `scale` setting adjusts the y-bounds and x-bounds so that the physical distance on one axis corresponds as closely as possible to physical distance on the other: for example, so that one centimetre will represent the same distance along each axis.

Normally each `LPGRAPH` statement produces one frame, and Genstat sets the size so that it will fill one screen or line-printer page, based on the settings of `WIDTH` and `PAGE` from `OPEN` or `OUTPUT` (3.3.1 and 3.4.3), or their defaults if these have not been specified. When output is going to a file the graph will be placed on a new page, unless this has been disabled using `OUTPUT`, `JOB` or `SET` (3.4.3, 5.1.1 and 5.6.1). The size of the graph is defined in terms of the number of characters in each row and the number of rows in the frame, a row being one line of output. You can adjust the size of the frame by using the `NROWS` and `NCOLUMNS` options; the minimum allowed is three rows and three columns, and the maximum number of columns is 17 characters less than the width of the output channel (to leave room for axis markings and titles). There is no maximum on the number of rows. By default, the number of columns is 101, subject to the maximum above, and the number of rows is the number of lines per page, less 8, to allow room for annotation. By defining the page size in advance you can avoid having to specify the numbers of rows and columns when you wish to plot many graphs.

The automatic axis scaling aims to find axis markings that are at reasonable values, but because the markings appear at fixed character positions this may not always be possible. If both upper and lower axis bounds are set, or `EQUAL` is set in conjunction with axis bounds, or you have requested integral axis markings, there may be conflicting constraints on the axis scaling. If the resultant axis markings then require several decimal places, you may be able to obtain better values by slight adjustments to the numbers of rows or columns.

The `MULTIPLE` option lets you generate several frames (separate graphs) from one statement. If there is room, the graphs can be printed alongside each other, for example to produce a two-by-two array of plots on a line-printer page. The option should be set to a variate whose elements define the number of graphs to plot in each frame and the number of values in the variate determines the number of frames to be output. For example,

```
    LPGRAPH [MULTIPLE=!(2,1,2)] A,B,C,D,E; X[1...3]
```

will produce three frames; the first containing `A` against `X[1]` and `B` against `X[2]`, the second containing `C` against `X[3]` and the third containing `D` against `X[1]` and `E` against `X[2]`. The sum of the values in the `MULTIPLE` list gives the total number of structures required to form the plots, which must therefore be equal to the length of the `Y` parameter list. The `X` list will be recycled if necessary, as here.

By default, each graph will fit the page (as if it had been produced by an individual `LPGRAPH` statement). However, if you set the `NCOLUMNS` option to a suitably small value, Genstat may be able to fit more than one frame across the page. The `MULTIPLE` option will then produce the graphs side by side. Remember that 17 columns are automatically added to provide annotation, and five blank columns are used to separate multiple graphs in parallel. This means that, for example, setting `NCOLUMNS=20` will produce two graphs in parallel on a screen of width 80, and three graphs when output to a file of width 121 or more.

You can annotate the graph by using the `TITLE`, `XTITLE` and `YTITLE` options described at the beginning of this section. If none of these are set, a simple key will be produced below the graph, as in Example 6.10.1a, which lists the identifiers and plotting symbols for each pair of `Y` and `X` structures. You can obtain your own key by setting the `DESCRIPTION` parameter, which supplies a line of text for each plot, as in Example 6.10.1b.

6.10.2 The `LPHISTOGRAM` directive

`LPHISTOGRAM` directive

Produces histograms using character (i.e. line-printer) graphics.

Options

`CHANNEL` = *scalar*	Channel number of output file; default is the current output file
`TITLE` = *text*	General title; default `*`
`LIMITS` = *variate*	Variate of group limits for classifying variates into groups; default `*`
`NGROUPS` = *scalar*	When `LIMITS` is not specified, this defines the number of groups into which a data variate is to be classified; default is the integer value nearest to the square root of the number of values in the variate
`LABELS` = *text*	Group labels
`SCALE` = *scalar*	Number of units represented by each character; default 1

Parameters

`DATA` = *identifiers*	Data for the histograms; these can be either a factor indicating the group to which each unit belongs, a variate whose values are to be grouped, or a one-way table giving the number of units in each group
`NOBSERVATIONS` = *tables*	One-way table to save numbers in the groups
`GROUPS` = *factors*	Factor to save groups defined from a variate
`SYMBOLS` = *texts*	Characters to be used to represent the bars of each histogram
`DESCRIPTION` = *texts*	Annotation for key

`LPHISTOGRAM` plots histograms or bar charts, depending on the input supplied by the `DATA` parameter: either a list of variates, a list of factors or a list of one-way tables. Histograms are formed from variates to provide quick and simple visual summaries of the data that they contain. The data values are divided into several groups, which are then displayed as a histogram consisting of a line of asterisks for each group. The number of asterisks in each line is proportional to the number of values assigned to that group; this figure is also printed at the beginning of each line.

Example 6.10.2a

```
   2  VARIATE Data
   3  READ Data

    Identifier   Minimum       Mean   Maximum     Values   Missing
          Data    0.0000      3.960     9.000         25         0

   5  LPHISTOGRAM Data

Histogram of Data
-----------------

                -   2  9 *********
              2 -   4  7 *******
              4 -   6  5 *****
              6 -   8  2 **
              8 -      2 **
```

```
Scale:  1 asterisk represents 1 unit.
```

You can specify a list of variates, to obtain a parallel histogram. For each group one row of asterisks is printed for each variate, labelled by the corresponding identifier.

As shown in Example 6.10.2b, the variates are sorted according to the same intervals. There is no need for them all to have the same numbers of values.

Example 6.10.2b

```
   6  VARIATE Data2
   7  READ Data2

    Identifier   Minimum      Mean   Maximum     Values   Missing
         Data2    0.0000     3.225     8.000         40         0

  10  LPHISTOGRAM Data,Data2

Histogram of Data and Data2
---------------------------


              - 1.5  Data   5 *****
                     Data2  9 *********

          1.5 - 3.0  Data   6 ******
                     Data2 14 **************

          3.0 - 4.5  Data   5 *****
                     Data2  7 *******

          4.5 - 6.0  Data   5 *****
                     Data2  8 ********

          6.0 - 7.5  Data   1 *
                     Data2  1 *

          7.5 -      Data   3 ***
                     Data2  1 *

Scale:  1 asterisk represents 1 unit.
```

You can use the `NGROUPS` option to specify the number of groups in the histogram; Genstat will then work out appropriate limits, based on the range of the data, to form intervals of equal width. For example:

```
LPHISTOGRAM [NGROUPS=5] Data
```

Alternatively, you can define the groups explicitly, by setting the `LIMITS` option to a variate containing the group limits. In Example 6.10.2c, `Limits` is a variate with seven values, producing a histogram in which the data is split into eight groups: ≤1, 1-2, 2-3, 3-5, 5-7, 7-8, 8-10, >10. The upper limit of each group is included within that group, so the group 3-5, for example, contains values that are greater than 3 and less than or equal to 5. The values of the limits variate are sorted into ascending order if necessary, but the variate itself is not changed.

Example 6.10.2c

```
  11  VARIATE [VALUES=1,2,3,5,7,8,10] Glimits
  12  LPHISTOGRAM [LIMITS=Glimits] Data
```

```
Histogram of Data grouped by Glimits
------------------------------------


              -  1.00  5 *****
        1.00 -  2.00  4 ****
        2.00 -  3.00  2 **
        3.00 -  5.00  7 *******
        5.00 -  7.00  4 ****
        7.00 -  8.00  1 *
        8.00 - 10.00  2 **
       10.00 -        0

Scale:  1 asterisk represents 1 unit.
```

You can use the LABELS option to provide your own labelling for the groups of the histogram. It should be set to a text vector of length equal to the number of groups. If neither NGROUPS nor LIMITS has been set, the number of groups is determined from the number of values in the LABELS structure. If LABELS is also unset, the default number of groups is chosen as the integer value nearest to the square root of the number of values (as in Example 6.10.2a where 25 values are sorted into five groups), up to a maximum of 10. Alternatively, procedure AKAIKEHISTOGRAM provides a more sophisticated method of generating histograms, using Akaike's Information Criterion (AIC) to generate an optimal grouping of the data.

If the DATA parameter is set to a factor or a one-way table, the histogram takes the form of a bar chart. There is now no longer the concept of dividing the x-axis into a set of contiguous intervals. Instead we have a set of bars located at various positions along the x-axis.

To form a bar chart from a factor, Genstat counts the number of units that occur with each level of the factor; thus the number of groups is the number of levels of the factor and the value for each group is the corresponding total. The labels of the factor (if present) are used to label the groups, as shown in Example 6.10.2d. Otherwise Genstat uses the factor levels.

Example 6.10.2d

```
  13  TEXT [VALUES=apple,banana,peach,cherry,pear,orange] Name
  14  FACTOR [LEVELS=6; LABELS=Name; NVALUES=32] Fruit
  15  READ Fruit

    Identifier    Values   Missing    Levels
         Fruit        32         0         6

  17  LPHISTOGRAM Fruit

Histogram of Fruit
------------------


 apple  3 ***
banana  2 **
 peach  8 ********
cherry  5 *****
  pear  8 ********
orange  6 ******


Scale:  1 asterisk represents 1 unit.
```

When Genstat plots the histogram of a one-way table, the number of groups is the number of levels of the factor classifying the table and the values of the table indicate the number of observations in each group. The labels or levels of the classifying factor are again used to label the histogram.

The `LABELS` option can also be used when producing a histogram from a factor or table. It should be set to a text of length equal to the number of levels of the factor or classifying factor.

When producing a parallel histogram the data structures must all be of the same type: variate, factor or table. Variates and factors may be restricted, in which case only the subset of values specified by the restriction will be included in the histogram; however, unlike many directives, restrictions do not carry over to the other structures listed by the `DATA` parameter. If parallel histograms are to be formed from several factors, they must all have the same number of levels, and the labels or levels of the first factor will be used to identify the groups. Likewise, if you are forming parallel histograms from several tables, they must all have the same number of values, and the classifying factor of the first table will define the labelling of the histogram.

The `SYMBOLS` parameter can specify alternative plotting characters to be used instead of the asterisk. For example:

```
LPHISTOGRAM Variate; SYMBOLS='+'
```

You can specify a different string for each structure in a parallel histogram. If you specify strings of more than one character, Genstat uses the characters in order, recycled as necessary, until each histogram bar is of the correct length.

Example 6.10.2e

```
  18  LPHISTOGRAM Data; SYMBOLS='X-O-'

Histogram of Data
-----------------


              -   2  9 X-O-X-O-X
            2 -   4  7 X-O-X-O
            4 -   6  5 X-O-X
            6 -   8  2 X-
            8 -      2 X-


Scale:  1 character represents 1 unit.
```

You can use the `DESCRIPTION` parameter to provide a text for labelling the histogram instead of the identifiers of the `DATA` structures.

Normally one asterisk will represent one unit. However, if there are many data values and the groups become large, Genstat may not be able to fit enough asterisks into one row. It will then alter the scaling so that one asterisk represents several units. You can set the scaling explicitly using the `SCALE` option; the value specified is rounded to the nearest integer, and determines how many units should be represented by each asterisk.

`LPHISTOGRAM` has two output parameters that allow you to save information that has been generated during formation of the histogram. The `NOBSERVATIONS` parameter allows you to save a one-way table of counts that contains the number of observations that were assigned to each group; the missing-value cell of this table will contain a count of the number of units that were missing and that therefore remain unclassified. When producing a histogram from a variate, you can use the `GROUPS` parameter to specify a factor to record the group to which each unit was allocated.

6.10.3 The LPCONTOUR directive

LPCONTOUR directive

Produces contour maps of two-way arrays of numbers using character (i.e. line-printer) graphics.

Options

CHANNEL = *scalar*	Channel number of output file; default is current output file
INTERVAL = *scalar*	Contour interval for scaling; default * i.e. determined automatically
TITLE = *text*	General title; default *
YTITLE = *text*	Title for y-axis; default *
XTITLE = *text*	Title for x-axis; default *
YLOWER = *scalar*	Lower bound for y-axis; default 0
YUPPER = *scalar*	Upper bound for y-axis; default 1
XLOWER = *scalar*	Lower bound for x-axis; default 0
XUPPER = *scalar*	Upper bound for x-axis; default 1
YINTEGER = *string token*	Whether y-labels integral (yes, no); default no
XINTEGER = *string token*	Whether x-labels integral (yes, no); default no
LOWERCUTOFF = *scalar*	Lower cut-off for array values; default *
UPPERCUTOFF = *scalar*	Upper cut-off for array values; default *

Parameters

GRID = *identifiers*	Pointers (of variates representing the columns of a data matrix), matrices or two-way tables specifying values on a regular grid
DESCRIPTION = *texts*	Annotation for key

A contour plot provides a way of displaying three-dimensional data in a two-dimensional plot. The data values are supplied as a rectangular array of numbers that represent the values of the variable in the third dimension, often referred to as *height* or the *z-axis*. The first two dimensions (x and y) are the rows and columns indexing the array; the complete three-dimensional data set is referred to as a *surface* or *grid*. Contours are lines that are used to join points of equal height, and usually some form of interpolation is used to estimate where these points lie. The resulting contour plot is not necessarily very "realistic" when compared to surface plots (6.4.3), but it has the advantage that the entire surface can easily be examined, without the danger of some parts being obscured by high points or regions.

You might use contour plots for example when you have data sampled at points on a regular grid, such as the concentrations of a trace element or nutrient in the soil. Contours are also very useful when fitting nonlinear models (2:3.8), when they can be used to study two-dimensional slices of the likelihood surface, to help find good initial estimates of the parameters.

LPCONTOUR produces output for a line printer by using cubic interpolation between the grid points to estimate a z-value for each character position in the plot. Each value is reduced to a single digit in the range 0 ... 9, according to the rules described below. To produce the contour plot only the even digits are printed: you can then see the contours as the boundaries between the blank areas and the printed digits.

In Example 6.10.3a, a function of two variables is calculated, and the shape of the function is displayed with LPCONTOUR. Titles have been given to the x-axis and the y-axis, and there is an overall title giving the algebraic form of the function.

Example 6.10.3a

```
MATRIX    [ROWS=5; COLUMNS=7] X,Y; VALUES=!((1...7)5),!(7(1...5))
CALCULATE Zvalues = (X-2.5)*(X-6)*X - 10*(Y-3)*(Y-3)
LPCONTOUR [TITLE='Z(x,y) = x*(x-2.5)*(x-6) - 10*(y-3)**2';\
          YTITLE='Y values'; XTITLE='X values'] Zvalues
```

```
Contour plot of Zvalues at intervals of 8.400

** Scaled values at grid points **
  -3.8690   -4.2857   -5.2976   -6.1905   -6.2500   -4.7619   -1.0119
  -0.2976   -0.7143   -1.7262   -2.6190   -2.6786   -1.1905    2.5595
   0.8929    0.4762   -0.5357   -1.4286   -1.4881    0.0000    3.7500
  -0.2976   -0.7143   -1.7262   -2.6190   -2.6786   -1.1905    2.5595
  -3.8690   -4.2857   -5.2976   -6.1905   -6.2500   -4.7619   -1.0119

                Z(x,y) = x*(x-2.5)*(x-6) - 10*(y-3)**2

          0.0000    0.1667    0.3333    0.5000    0.6667    0.8333    1.0000
           '         '         '         '         '         '         '

     1.000-66666             4444444444               4444444   666  888-
           666666666666666          44444444444444444444      666   88
                       6666666666                         66666   88   0
           8888888            6666666666              666666    888  00
           88888888888888          6666666666666666666666     888   00
                   88888888888           666666666666       8888   00  2
     0.750-              888888888                        8888   000  22-
           0000             8888888888                 888888   000   22
Y          0000000000          88888888888          8888888    000   22
           0000000000000         8888888888888888888888888     000  222
v          00000000000000          8888888888888888888888     000   22
a          000000000000000          888888888888888888888     000   22
l    0.500-0000000000000000         88888888888888888888     0000   22  -
u          000000000000000          888888888888888888888     000   22
e          00000000000000          8888888888888888888888     000   22
s          0000000000000         8888888888888888888888888     000  222
           0000000000          88888888888          8888888    000   22
           0000             8888888888                 888888   000   22
     0.250-              888888888                        8888   000  22-
                   88888888888           666666666666       8888   00  2
           88888888888888          6666666666666666666666     888   00
           8888888            6666666666              666666    888  00
                       6666666666                         66666   88   0
           666666666666666          44444444444444444444      666   88
     0.000-66666             4444444444               4444444   666  888-
           '         '         '         '         '         '         '

                                     X values
```

The `GRID` parameter can be set to a matrix, a two-way table (with the first factor defining the rows), or a pointer to a set of variates each containing a column of data. We explain the conventions in terms of a matrix as input, but similar rules apply to the other structures. When reading or printing a matrix the origin of the rows and columns (row 1, column 1) appears at the top left-hand corner. However, in forming the contour plot the rows are reversed in order so that the first row of the matrix is placed at the bottom of the contour; thus the origin of the contour is located, according to the usual conventions, at the bottom left-hand corner of the plot. (By default, the `DCONTOUR` directive reverses the rows of the grid in the same way, but it also has an `ORIENTATION` option that allows you to plot with the normal orientation; see 6.4.1.)

`LPCONTOUR` scales the grid values by dividing by the contour interval. The scaled grid values are then converted to single digits by taking the remainder modulo 10 and truncating the fractional part. In Example 6.10.3a, the first grid value is −32.5, which is divided by the interval size (8.4) to obtain −3.869; this becomes 6.131 when taken modulo 10, and then 6 after truncation. To aid interpretation of the plot, the array of scaled values is printed out.

The `INTERVAL` option allows you to set the interval between contour lines. For example, if the grid values range from 17 to 72 and the interval is set to 10, contour lines (the boundaries between blank space and printed digits) will occur at grid values of 20, 30, 40, 50, 60 and 70. By default, the interval is determined from the range of the data in order to obtain 10 contours.

The `UPPERCUTOFF` and `LOWERCUTOFF` options can be used to define a window for the grid values that will form the contours. All values above or below these are printed as `X`. Setting either `UPPERCUTOFF` or `LOWERCUTOFF` will change the default contour interval, as the range of data values is effectively curtailed.

You can use the `TITLE`, `YTITLE` and `XTITLE` options to annotate the contour plot. If you specify several grids, these will be plotted in separate frames and the text of the `TITLE` option will appear at the top of each one. You should thus use `TITLE` only to give a general description of what the contours represent. The `DESCRIPTION` parameter can be used to add specific descriptions to be printed at the bottom of each individual plot.

The `YUPPER` and `YLOWER` options allow you to set upper and lower bounds for the y-axis; thus generating axis labels that reflect the range of values over which the grid was observed or evaluated. Setting `YINTEGER=yes` will ensure the labels are printed as integers, if possible. The default axis bounds are 0.0 and 1.0. The options `XLOWER`, `XUPPER` and `XINTEGER` similarly control labelling of the x-axis.

Example 6.10.3b shows how a contour plot can be produced from a set of variates. In line 22, the values of the variates are inverted, using the `REVERSE` function, and y-axis labelling set up so that *depth* increases as you read down the plot. (The same data are plotted in Figure 6.4.1a using the `DCONTOUR` directive, but there the `YORIENTATION` option is used to reverse the y-axis.)

Example 6.10.3b

```
  6  " Core samples were taken from a wetland rice experiment to examine
 -7    the leaching of ammonium nitrate. Three cores were taken at
 -8    intervals of 5cm, and the concentration of ammonium nitrate was
 -9    measured at depths of 4, 8, ... 20 cm. "
 10  VARIATE   [NVALUES=5] Core[1...5]
 11  READ      Core[]

   Identifier   Minimum      Mean   Maximum    Values   Missing
      Core[1]     5.000     8.200     11.00         5         0
      Core[2]     6.000     67.60     195.0         5         0
      Core[3]     129.0     940.6      2315         5         0
      Core[4]     10.00     36.00     77.00         5         0
      Core[5]     7.000     9.400     15.00         5         0

 17  TEXT      [VALUES=' Samples taken 40 days after placement ', \
 18            ' of 2 grams supergranule urea. '] Coredesc
 19  CALCULATE Core[] = LOG10(REVERSE(Core[]))
 20  LPCONTOUR [YTITLE='Soil depth in cm';  \
 21            XTITLE='Distance from central core'; \
 22            YINTEGER=yes; XINTEGER=yes; \
 23            YUPPER=4; YLOWER=20; XUPPER=10; XLOWER=-10] \
 24            Core; DESCRIPTION=Coredesc

 Contour plot of Core at intervals of 0.267

 ** Scaled values at grid points **
    3.1704    2.9193    7.9179    3.7515    3.5799
    3.3880    7.1797   12.6222    6.4687    3.1704
    2.6222    8.5911   11.3557    7.0772    3.1704
    3.7515    5.7926   11.3091    5.5415    3.5799
    3.9068    4.8808    8.2790    3.7515    4.4121
```

```
        -10        -5         0         5        10
           '         '         '         '         '
         4- 2222222222  4  66      66  44           -
              222     44 66  88 888  66 444
                   444  6  88  0   8  66 4444
                 444  66 88  00 00  88 66  444
S              4444  66 88 00    00  8  6   444
o             444   66 88 00   2  00  8 66  4444
i        8-  444   66  8  0  2222  00 8  66  444    -
l            444  66  88 00  2222  00 88 66   44
             44  66  88  00  2222  00 88 666  444
d          2 44  66  88 000   222  00 88  66  444
e          2 44  6  88  000       000 88  66   44
p          2 44 66  88  000       00  88  66   44
t       12-2 44 66  88   000     000  88  66   44   -
h          2 44  66  88  0000    000  88 666  444
           2 44  666  88  000    00  88  66   444
i             44  666  88  00    00  88  66  4444
n             444  666 88  000   00  88 66   444
              4444   66 88  00   00  8  6   4444
c       16-    4444   6  8  00  000 88 66  4444     -
m              44444  66 88  00 00  8  6  4444
                44444  6  88  000  88 66 4444
                44444  66 88   0  88  6  444
                444444 66  88    88  66 444        4
                444444  66  888888  66  44         4
        20-    4444444  666   88   66  44         44-
           '         '         '         '         '

                  Distance from central core

       Samples taken 40 days after placement
           of 2 grams supergranule urea.
```

7 Summary of other facilities

In this chapter we summarise the other facilities in Genstat. Many of these are covered in Part 2 of the *Guide to the Genstat Command Language*. Details of the commands not covered there can be found in Part 2 of the *Genstat Reference Manual* (for directives), or Part 3 of the *Manual* (for procedures).

7.1 Basic statistics

Genstat provides a wide range of commands for basic statistics and exploratory analysis. Some are available through specially-designed commands (usually procedures in the Genstat Procedure Library). Others simply use basic options of more powerful commands (usually directives).

`DESCRIBE`	saves and/or prints summary statistics for variates (2:2.1.1)
`TALLY`	forms a simple tally table of the distinct values in a vector (2:2.2.5)
`CDESCRIBE`	calculates summary statistics and tests of circular data (2:2.1.2)
`CASSOCIATION`	calculates measures of association for circular data
`CCOMPARE`	tests whether samples from circular distributions have a common mean direction or have identical distributions
`FCORRELATION`	forms correlations between variates, and calculates their probabilities (2:2.8.1)
`BLANDALTMAN`	produces Bland-Altman plots to assess the agreement between two variates (2:2.8.8)
`BOXPLOT`	draws box-and-whisker diagrams or schematic plots (2:2.2.2)
`DOTPLOT`	produces a dot-plot (2:2.2.6)
`RUGPLOT`	draws "rugplots" to display the distribution of one or more samples (2:2.2.3)
`STEM`	produces a simple stem-and-leaf chart (2:2.2.4)
`TTEST`	performs a one- or two-sample t-test (2:2.3.1)
`AONEWAY`	provides one-way analysis of variance (2:2.3.2)
`A2WAY`	performs analysis of variance of a balanced or unbalanced design with up to two treatment factors (2:2.3.3)
`A2DISPLAY`	provides further output from an `A2WAY` analysis (2:2.3.3)
`A2KEEP`	saves information from an `A2WAY` analysis (2:2.3.3)
`CHIPERMTEST`	does a random permutation test for a two-dimensional contingency table (2:2.9.2)
`CHISQUARE`	calculates chi-square statistics for one- and two-way tables (2:2.9.1)
`CMHTEST`	performs the Cochran-Mantel-Haenszel test (2:2.9.5)
`EDFTEST`	performs empirical-distribution-function goodness-of-fit tests (2:2.2.12)
`FEXACT2X2`	does Fisher's exact test for 2×2 tables (2:2.9.2)
`FRIEDMAN`	performs Friedman's nonparametric analysis of variance (2:2.6.2)
`STEEL`	performs Steel's many-one rank test (2:2.6.3)
`TEQUIVALENCE`	performs equivalence, non-inferiority and non-superiority tests
`BNTEST`	calculates one- and two-sample binomial tests (2:2.3.4)

PNTEST	calculates one- and two-sample Poisson tests (2:2.3.5)
GSTATISTIC	calculates the gamma statistic of agreement for ordinal data (2:2.8.6)
HCOMPAREGROUPINGS	calculates the Rand index, adjusted Rand index or Jaccard index to compare groupings defined by two factors (2:6.19.7)
KAPPA	calculates a kappa coefficient of agreement for nominally scaled data (2:2.8.5)
KCONCORDANCE	calculates Kendall's Coefficient of Concordance, synonym CONCORD (2:2.8.4)
KOLMOG2	performs a Kolmogorov-Smirnoff two-sample test (2:2.5.2)
KRUSKAL	carries out a Kruskal-Wallis one-way analysis of variance (2:2.6.1)
KTAU	calculates Kendall's rank correlation coefficient τ (2:2.8.3)
LCONCORDANCE	calculates Lin's concordance correlation coefficient (2:2.8.7)
MANNWHITNEY	performs a Mann-Whitney U test (2:2.5.1)
MCNEMAR	performs McNemar's test for the significance of changes (2:2.9.3)
QCOCHRAN	performs Cochran's *Q* test for differences between related samples (2:2.9.4)
RUNTEST	performs a test of randomness of a sequence of observations (2:2.4.3)
SIGNTEST	performs a one or two sample sign test (2:2.4.2)
SPEARMAN	calculates Spearman's rank correlation coefficient (2:2.8.2)
WILCOXON	performs a Wilcoxon Matched-Pairs (Signed-Rank) test (2:2.4.1)

There are also commands for studying distributions of samples of data:

DISTRIBUTION	estimates the parameters of continuous and discrete distributions (2:2.2.10)
DPROBABILITY	plots probability distributions, and estimates their parameters (2:2.2.7)
FDRMIXTURE	estimates false discovery rates using mixture distributions
KERNELDENSITY	uses kernel density estimation to estimate a sample density (2:2.2.8)
NORMTEST	performs tests of univariate and/or multivariate Normality (2:2.2.11)
WSTATISTIC	calculates the Shapiro-Wilk test for Normality (2:2.2.11)

7.2 Regression analysis

Genstat provides directives for carrying out linear and nonlinear regression, also generalized linear, generalized additive and generalized nonlinear models. They are designed to allow easy comparison between models, and comparison between groups of data (specified as factors). The directives for nonlinear regression can also be used for general optimization. There are three preliminary directives for defining the form of model to be fitted, of which the MODEL directive must always be given first:

MODEL	defines the response variate(s) and the type of model to be fitted (2:3.1.1)
TERMS	specifies a maximal model, containing all terms to be used

	in subsequent regression models (2:3.2.3)
`RCYCLE`	controls iterative fitting of generalized linear models, generalized additive models and nonlinear models, and specifies parameters and bounds for nonlinear models (2:3.5.4)

Separate directives carry out the fitting of the various types of model:

`FIT`	fits a linear model, a generalized linear model, a generalized additive model, or a generalized nonlinear model (2:3.1.2)
`FITCURVE`	fits a standard nonlinear regression model (2:3.7.1)
`FITNONLINEAR`	fits a user-defined nonlinear regression model or optimizes a scalar function (2:3.8.2)

Further directives are provided to allow sequential modification of the set of explanatory variables:

`ADD`	adds extra terms to any type of regression model (2:3.2.4)
`DROP`	drops terms from any type of regression model (2:3.2.4)
`SWITCH`	adds terms to, or drops them from, any type of regression model (2:3.2.4)
`TRY`	displays results of single-term changes to a linear or generalized linear model (2:3.2.5)
`STEP`	selects terms to include in or exclude from a linear or generalized linear model (2:3.2.7)

Once you have fitted the model, you can display further results, form and compare predictions, plot the fitted model, produce diagnostic plots, store the results in data structures for use elsewhere in Genstat, do permutation (or exact) texts, or calculate power information about the model:

`RDISPLAY`	displays the fit of any type of regression model (2:3.1.3, 2:3.5.3, 2:3.7.4)
`PREDICT`	forms predictions from a linear or generalized linear model (2:3.3.4, 2:3.5.3)
`RCOMPARISONS`	calculates comparison contrasts amongst regression means (2:3.3.5)
`RCURVECOMMONNONLINEAR`	refits a standard curve with common nonlinear parameters across groups to provide s.e.'s for linear parameters (2:3.7.7)
`RFUNCTION`	estimates functions of parameters of any type of regression model (2:3.7.5)
`RGRAPH`	draws a graph to display the fit of any type of regression model (2:3.1.6)
`RCHECK`	provides diagnostic plots and other information for checking the fit of any type of regression model (2:3.1.7)
`RDESTIMATES`	plots one- or two-way tables of regression estimates (2:3.3.8)
`RKEEP`	stores the results from any type of regression model (2:3.1.4, 2:3.5.3, 2:3.7.4)
`RSPREADSHEET`	puts results from a regression, generalized linear or nonlinear model into spreadsheets (2:3.1.5)
`RKESTIMATES`	saves estimates and other information about individual terms in a regression analysis (2:3.2.2)
`RWALD`	calculates Wald and F tests for dropping terms from a

	regression (2:3.2.6)
`RPERMTEST`	does random permutation tests for regression models (2:3.1.9)
`RPOWER`	calculates the power (probability of detection) for regression models (2:3.1.8)

There are also many specialized procedures in the `regression` and `glm` modules of the Library; see Part 3 of the *Genstat Reference Manual*.

`BREGRESSION`	constructs a regression tree (2:3.9.1)
`BRDISPLAY`	displays a regression key (2:3.9.2)
`BRVALUES`	forms values for nodes of a regression tree (2:3.9.3)
`BPRUNE`	prunes a tree using minimal cost complexity (4.12.8, 2:3.9.3)
`BRPREDICT`	makes predictions using a regression tree (2:3.9.4)
`BRKEEP`	saves information from a regression tree (2:3.9.5)
`BRFOREST`	constructs a random regression forest
`BRFDISPLAY`	displays information about a random regression forest
`BRFPREDICT`	makes predictions using a random regression forest
`FITINDIVIDUALLY`	fits regression and generalized linear models one term at a time (2:3.5.3)
`GEE`	fits models to longitudinal data by generalized estimating equations (2:3.5.12)
`GLM`	analyses non-standard generalized linear models
`GLMM`	fits a generalized linear mixed model (2:3.5.10)
`GLDISPLAY`	displays further output from a `GLMM` analysis (2:3.5.10)
`GLKEEP`	saves results from a `GLMM` analysis (2:3.5.10)
`GLPERMTEST`	does random permutation tests for generalized linear mixed models (2:3.5.10)
`GLPLOT`	plots residuals from a `GLMM` analysis (2:3.5.10)
`GLPREDICT`	forms predictions from a `GLMM` analysis (2:3.5.10)
`HGANALYSE`	analyses data using hierarchical generalized linear models (2:3.5.11)
`HGDISPLAY`	displays a hierarchical generalized linear model analysis (2:3.5.11)
`HGFIXEDMODEL`	defines the fixed model for a hierarchical generalized linear model (2:3.5.11)
`HGFTEST`	calculates likelihood tests for fixed terms in a hierarchical generalized linear model (2:3.5.11)
`HGKEEP`	saves information from a hierarchical generalized linear model analysis (2:3.5.11)
`HGNONLINEAR`	defines nonlinear parameters for the fixed model of a hierarchical generalized linear model (2:3.5.11)
`HGPLOT`	produces model-checking plots for a hierarchical generalized linear model analysis (2:3.5.11)
`HGGRAPH`	draws a graph to display the fit of hierarchical generalized linear model analysis (2:3.5.11)
`HGPREDICT`	forms predictions from hierarchical generalized linear model analysis (2:3.5.11)
`HGRANDOMMODEL`	defines the random model for a hierarchical generalized linear model (2:3.5.11)
`HGDRANDOMMODEL`	extends a hierarchical generalized linear model to become a double hierarchical generalized linear model (2:3.5.11)

HGRTEST	calculates likelihood tests for random terms in a hierarchical generalized linear model (2:3.5.11)
HGSTATUS	displays the current HGLM model definitions (2:3.5.11)
HGWALD	Prints or saves Wald tests for fixed terms in an HGLM (2:3.5.11)
PROBITANALYSIS	fits probit models allowing for natural mortality and immunity (2:3.5.9)
FIELLER	calculates effective doses and relative potencies (2:3.5)
MICHAELISMENTEN	fits the Michaelis-Menten equation for substrate concentration versus time data
MMPREDICT	predicts the Michaelis-Menten curve for a particular set of parameter values
NLAR1	fits curves with an AR1 or a power-distance correlation model (2:8.1.6)
RAR1	fits regressions with an AR1 or a power-distance correlation model (2:8.1.6)
RQLINEAR	fits and plots quantile regressions for linear models (2:3.10.1)
RQNONLINEAR	fits and plots quantile regressions for nonlinear models
RQSMOOTH	fits and plots quantile regressions for loess or spline models
RSCREEN	performs screening tests for generalized or multivariate linear models (2:3.2.9)
RSEARCH	helps search through models for a regression or generalized linear model (2:3.2.8)
R0INFLATED	fits zero-inflated regression models to count data with excess zeros (2:3.5.13)
R0KEEP	saves information from models fitted by R0INFLATED (2:3.5.13)
RBRADLEYTERRY	fits the Bradley-Terry model for paired-comparison preference tests
RCATENELSON	performs a Cate-Nelson graphical analysis of bivariate data
RCIRCULAR	does circular regression of mean direction for an angular response
RFINLAYWILKINSON	performs Finlay and Wilkinson's joint regression analysis of genotype-by-environment data
RIDGE	does ridge regression and principal component regression analyses
LRIDGE	does logistic ridge regression
RLASSO	performs lasso using iteratively reweighted least-squares
RLFUNCTIONAL	fits a linear functional relationship model
RMGLM	fits a model where different units follow different generalized linear models
RNEGBINOMIAL	fits a negative binomial generalized linear model, estimating the aggregation parameter
RNONNEGATIVE	fits a generalized linear model with nonnegativity constraints
RPAIR	gives t-tests for all pairwise differences of means from linear or generalized linear models
RPARALLEL	carries out analysis of parallelism for nonlinear functions

RQUADRATIC	fits a quadratic surface and estimates its stationary point
RRETRIEVE	retrieves a regression save structure from an external file
RSTORE	stores a regression save structure in an external file
RSCHNUTE	fits a general four-parameter growth model to a non-decreasing response variate
RYPARALLEL	fits the same regression model to several response variates, and collates the output
R2LINES	fits two-straight-line (broken-stick) models
IFUNCTION	estimates implicit and/or explicit functions of parameters
MINIMIZE	finds the minimum of a function calculated by a procedure
MIN1DIMENSION	finds the minimum of a function in one dimension
SIMPLEX	searches for the minimum of a function using the Nelder-Mead simplex algorithm
SVGLM	fits generalized linear models to survey data
YTRANSFORM	estimates the parameter lambda of a single parameter transformation
XOCATEGORIES	performs analyses of categorical data from cross-over trials
EXTRABINOMIAL	fits models to overdispersed proportions
DILUTION	calculates most probable numbers from dilution series data
DSEPARATIONPLOT	creates a separation plot for visualising the fit of a model with a dichotomous (i.e. binary) or polytomous (i.e. multi-categorical) outcome
WADLEY	fits models for Wadley's problem, allowing alternative links and errors

7.3 Analysis of variance

Genstat has a very general algorithm for analysis of variance of balanced experiments. There are several directives to define the various aspects of model to be fitted:

BLOCKSTRUCTURE	defines the blocking structure of the design, and hence the strata and error terms (2:4.2.1)
COVARIATE	specifies a list of covariates for analysis of covariance (2:4.3.1)
TREATMENTSTRUCTURE	defines the treatment (or systematic) terms (2:4.1.1)

For unstructured designs with a single error term, BLOCKSTRUCTURE need not be specified, and COVARIATE is needed only for analysis of covariance. Alternatively, the AFCOVARIATES procedure allows more complicated types of covarate model to be defined:

AFCOVARIATES	specifies covariates from a model formula for analysis of covariance (2:4.3.2)

Once the model has been defined, the y-variates can be analysed using the ANOVA directive:

ANOVA	performs analysis of variance (2:4.1.2)

Then, after you have fitted the model, you can display or calculate further results, plot means and residuals, check assumptions or store the results in data structures for use elsewhere in Genstat:

ADISPLAY	displays further output from analyses produced by ANOVA (2:4.1.3)
AGRAPH	plots tables of means from ANOVA (2:4.1.5)
APLOT	plots residuals from an ANOVA analysis (2:4.1.4)
AFIELDRESIDUALS	display residuals from a field experiment in field layout

	(2:4.1.4)
ABLUPS	calculates BLUPs for block terms in an ANOVA analysis (2:4.2.2)
ACHECK	checks the assumptions for an ANOVA analysis (2:4.1.6)
AKEEP	copies information from an ANOVA analysis into Genstat data structures (2:4.6.1)
APERMTEST	performs random permutation and exact tests for analysis of variance (2:4.1.7)
APOLYNOMIAL	calculates the equation for a polynomial contrast fitted by ANOVA (2:4.5.1)
ADPOLYNOMIAL	plots single-factor polynomial contrasts fitted by ANOVA (2:4.5.2)
ARESULTSUMMARY	provides a summary of results from an ANOVA analysis (2:4.1.3)
ASPREADSHEET	saves analysis of variance results in a spreadsheet (2:4.6.3)
ASTATUS	provides information about the settings of ANOVA models and variates (2:4.9.1)
AMCOMPARISON	performs pairwise multiple comparison tests for ANOVA means (2:4.1.9)
AMDUNNETT	forms Dunnett's simultaneous confidence interval around a control (2:4.1.10)
ACONFIDENCE	calculates simultaneous confidence intervals (2:4.1.8)
FALIASTERMS	forms information about aliased model terms in analysis of variance

The designs analysed by ANOVA are said to be *balanced* or, more accurately, to have the property of *first-order balance* (see 2:4.7). They include virtually all the standard experimental designs, and ANOVA itself detects if the necessary conditions are not met.

Unbalanced designs with a single error term can be analysed using procedures AUNBALANCED, AUDISPLAY and AUKEEP. The model is specified just as for ANOVA but the analysis uses the Genstat regression facilities. If you have only two treatment factors in an unbalanced design with a single error term, it may be more convenient to use A2WAY. Unbalanced designs with several error terms can be analysed by the REML directive (7.5). However, if the additional random terms contain very little information about the treatments, it may be more convenient (and equally effective) to treat these as fixed nuisance terms, and use AUNBALANCED. Decisions like this can be made using the AOVANYHOW procedure.

AUNBALANCED	performs analysis of variance for unbalanced designs (2:4.8.1)
AUDISPLAY	produces further output for an unbalanced design (2:4.8.2)
AUGRAPH	plots tables of means from an unbalanced design (2:4.8.3)
AUKEEP	saves output from analysis of an unbalanced design (2:4.8.4)
AUPREDICT	forms predictions from an unbalanced design (2:4.8.5)
AUSPREADSHEET	Saves results from an analysis of an unbalanced design in a spreadsheet (2:4.8.6)
AUMCOMPARISON	performs pairwise multiple comparison tests for means from unbalanced analysis of variance
A2WAY	performs analysis of variance of a balanced or unbalanced design with up to two treatment factors (2:2.3.3)
A2DISPLAY	provides further output following an analysis of variance by A2WAY (2:2.3.3)
A2KEEP	copies information from an A2WAY analysis into Genstat

	data structures (2:2.3.3)
A2RESULTSUMMARY	provides a summary of results from an analysis by A2WAY (2:2.3.3)
AN1ADVICE	aims to give useful advice if a design that is thought to be balanced fails to be analysed by ANOVA (2:4.8.8)
AOVANYHOW	performs analysis of variance using ANOVA, AUNBALANCED, A2WAY or REML as appropriate (2:4.8.7)
AOVDISPLAY	provides further output from an analysis by AOVANYHOW

Other procedures relevant to analysis of variance, in the aov module of the Library, include:

AMTIER	analyses a multitiered design specified by up to three model formulae (2:4.2.3)
AMTDISPLAY	displays further output for multitiered designs (2:4.2.3)
AMTKEEP	saves information from the analysis of a multitiered design by AMTIER (2:4.2.3)
VSPECTRALCHECK	forms the spectral components from the canonical components of a multitiered design, and constrains any negative spectral components to zero
AONEWAY	provides one-way analysis of variance (2:2.3.2)
ASCREEN	performs screening tests for designs with orthogonal block structure (2:4.7.6)
ABIVARIATE	produces graphs and statistics for bivariate analysis of variance
ABOXCOX	estimates the power λ in a Box-Cox transformation, that maximizes the partial log-likelihood in ANOVA
ACANONICAL	determines the orthogonal decomposition of the sample space for a design, using an analysis of the canonical relationships between the projectors derived from two or more model formulae.
ACDISPLAY	provides further output from an analysis by ACANONICAL.
ACKEEP	saves information from an analysis by ACANONICAL.
AFMEANS	forms tables of means classified by ANOVA treatment factors (2:4.1.5)
AMMI	provides exploratory analysis of genotype × environment interactions
FMEGAENVIRONMENTS	forms mega-environments based on winning genotypes from an AMMI-2 model
APAPADAKIS	analysis of variance with an added Papadakis covariate, formed from neighbouring residuals
AREPMEASURES	produces an analysis of variance for repeated measurements (2:8.1.3)
ARETRIEVE	retrieves an ANOVA save structure from an external file
ASTORE	stores an ANOVA save structure in an external file
AYPARALLEL	does the same analysis of variance for several y-variates, and collates the output
A2PLOT	plots effects from two-level designs with robust s.e. estimates
A2RDA	saves results from an analysis of variance in R data frames
AU2RDA	saves results from an unbalanced analysis of variance, by AUNBALANCED, in R data frames
CENSOR	pre-processes censored data before analysis by ANOVA
CINTERACTION	clusters rows and columns of a two-way interaction table

DIALLEL	analyses full and half diallel tables with parents
FRIEDMAN	performs Friedman's nonparametric analysis of variance (2:2.6.2)
LVARMODEL	analyses a field trial using the Linear Variance Neighbour model
NLCONTRASTS	fits non-linear contrasts to quantitative factors in ANOVA
SED2ESE	calculates effective standard errors that give good approximate standard errors of differences
SEDLSI	calculates least significant intervals
LSIPLOT	plots least significant intervals, saved from SEDLSI
TEQUIVALENCE	performs equivalence, non-inferiority and non-superiority tests
VHOMOGENEITY	tests homogeneity of variances
WSTATISTIC	calculates the Shapiro-Wilk test for Normality

Full details can be found in Part 3 of the *Genstat Reference Manual*.

7.4 Design of experiments

Genstat has a comprehensive set of facilities for design of experiments ranging from procedures that allow you to select and generate a design from an extensive repertoire of possibilities, to directives and procedures that enable you to develop new designs and assess their properties. Collectively, these are known as the *Genstat Design System*. Many different design types are covered, each with a procedure that allows you to view and choose from the available possibilities. Other procedures allow designs and data forms to be displayed. There is also a general procedure DESIGN that can be used interactively to provide a single point of access to all the design types.

DESIGN	provides a menu-driven interface for selecting and generating experimental designs (2:4.9.1)
AGALPHA	forms alpha designs for up to 100 treatments (2:4.9.7)
AGBIB	generates balanced-incomplete-block designs (2:4.9.8)
AGBOXBEHNKEN	generates Box-Behnken designs (2:4.9.12)
AGCENTRALCOMPOSITE	generates central composite designs (2:4.9.11)
AGCROSSOVERLATIN	generates Latin squares balanced for carry-over effects (2:4.9.3)
AGCYCLIC	generates cyclic designs from standard generators (2:4.9.9)
AGDESIGN	generates generally balanced designs – factorial designs with blocking, fractional factorial designs, Lattice squares etc. (2:4.9.3)
AGFACTORIAL	generates minimum aberration block or fractional factorial designs (2:4.9.2)
AGFRACTION	generates fractional factorial designs
AGHIERARCHICAL	generates orthogonal hierarchical designs (2:4.9.1)
AGLATIN	generates mutually orthogonal Latin squares (2:4.9.4)
AGLOOP	generates loop designs e.g. for time-course microarray experiments (2:4.9.17)
AGMAINEFFECT	generates designs to estimate main effects of two-level factors (2:4.9.13)
AGNEIGHBOUR	generates neighbour-balanced designs (2:4.9.10)
AGNONORTHOGONALDESIGN	generates non-orthogonal multi-stratum designs
AGSPACEFILLINGDESIGN	generates space filling designs
AGQLATIN	generates complete and quasi-complete Latin squares

	(2:4.9.3)
`AGREFERENCE`	generates reference-level designs e.g. for microarray experiments (2:4.9.16)
`AGSEMILATIN`	generates semi-Latin squares (2:4.9.5)
`AGSQLATTICE`	generates square lattice designs (2:4.9.6)
`PDESIGN`	prints treatment combinations tabulated by the block factors (2:4.10.1)
`DDESIGN`	plots the plan of a design (2:4.10.2)
`ADSPREADSHEET`	puts the data and plan of an experimental design into Genstat spreadsheets (2:4.10.3)

`DESIGN` and the `AG`... procedures (above) that it calls provide the Select Design facilities in Genstat *for Windows*, while the alternative Standard Design menu uses `AGHIERARCHICAL`, `AGLATIN` and `AGSQLATTICE` to generate completely randomized designs, randomized blocks, Latin and Graeco-Latin squares, split-plots, strip-plots (or criss-cross designs) and lattices.

There are also procedures that you can use to determine the sample size (i.e. replication) required for experiments that are to be analysed by analysis of variance, t-test or various non-parametric tests. You can also calculate the power (or probability of detection) for terms in analysis of variance or regression analyses.

`APOWER`	calculates the power (probability of detection) for terms in an analysis of variance (2:4.12.3)
`ASAMPLESIZE`	finds the replication (sample size) to detect a treatment effect or contrast (2:4.12.2)
`RPOWER`	calculates the power (probability of detection) for regression models (2:3.1.8)
`ADETECTION`	calculates the minimum size of effect or contrast detectable in an analysis of variance (2:4.12.4)
`SBNTEST`	calculates the sample size for binomial tests (2:4.12.5)
`SCORRELATION`	calculates the sample size to detect specified correlations (2:4.12.10)
`SLCONCORDANCE`	calculates the sample size for Lin's concordance coefficient (2:4.12.11)
`SMANNWHITNEY`	calculates the sample size for the Mann-Whitney test (2:4.12.9)
`SMCNEMAR`	calculates the sample size for McNemar's test (2:4.12.8)
`SPNTEST`	calculates the sample size for a Poisson test (2:4.12.6)
`SPRECISION`	calculates the sample size to obtain a specified precision
`SSIGNTEST`	calculates the sample size for a sign test (2:4.12.7)
`STTEST`	calculates the sample size for t-tests, including equivalence tests and tests for non-inferiority (2:4.12.1)
`DSTTEST`	plots power and significance for t-tests, including equivalence tests

The design-generation procedures form and randomize the designs automatically, calling other directives and procedures to perform the necessary tasks, and there is no need for you to be aware of any of the details. However, we give more information in Sections 4.8 - 4.10 and 4.12 of Part 2, in case you want to study the process in more depth or to add new designs. Briefly, the design system is based on a range of standard generators. Some of these, such as the Galois fields used to generate Latin squares or the Hadamard matrices needed for main-effect designs, can be formed when required – and so there is no limitation on the available designs. Repertoires of others, such as design keys, are stored in backing-store files which are scanned by the design generation procedures to form menus listing the available possibilities. Algorithms are available to form generators for new designs, and these can then be added to the design files to become an integral part of the system. Other design utilities include procedures for combining simple designs into more complicated arrangements, for constructing augmented designs, and for determining how many replicates are needed. There are also directives for constructing response-surface designs using the BLKL algorithm of Atkinson & Donev (1992) and for constructing doubly resolvable row-column designs. The relevant commands include the directives

`AFMINABERRATION`	forms minimum aberration factorial or fractional-factorial designs
`AFRESPONSESURFACE`	uses the BLKL algorithm to construct designs for estimating response surfaces (2:4.9.14)
`AGRCRESOLVABLE`	forms doubly resolvable row-column designs
`GENERATE`	generates values of factors in systematic order or as defined by a design key, or forms values of pseudo-factors (2:4.13.1)
`RANDOMIZE`	puts units of vectors into random order, or randomizes units of an experimental design (2:4.11.1)
`FKEY`	forms design keys for multi-stratum experimental designs, allowing for confounding and aliasing of treatments (2:4.13.6)
`FPSEUDOFACTORS`	determines patterns of confounding and aliasing from design keys, and extends the treatment formula to incorporate the necessary pseudo-factors (2:4.13.7)
`SET2FORMULA`	forms a model formula using structures supplied in a pointer (4.8.3)

and the procedures

`AEFFICIENCY`	calculates efficiency factors for experimental designs
`AFALPHA`	generates alpha designs (2:4.9.6)
`AFAUGMENTED`	forms an augmented design (2:4.13.5)
`AFCARRYOVER`	forms factors to represent carry-over effects in cross-over trials
`AFCYCLIC`	generates block and treatment factors for cyclic designs (2:4.9.8)
`AFLABELS`	forms a variate of unit labels for a design
`AFNONLINEAR`	forms D-optimal designs to estimate the parameters of a nonlinear or generalized linear model (2:4.9.15)
`AFPREP`	searches for an efficient partially-replicated design
`AFRCRESOLVABLE`	forms doubly resolvable row-column designs, with output
`AFUNITS`	forms a factor to index the units of the final stratum of a design
`AKEY`	generates values for treatment factors using the design key method (2:4.13.2)
`AMERGE`	merges extra units into an experimental design (2:4.13.3)
`APRODUCT`	forms a new experimental design from the product of two designs (2:4.13.4)
`ARANDOMIZE`	randomizes and prints an experimental design (2:4.11.2)
`COVDESIGN`	produces experimental designs efficient under analysis of covariance
`FACDIVIDE`	represents a factor by factorial combinations of a set of factors
`FACPRODUCT`	forms a factor with a level for every combination of other

	factors
FBASICCONTRASTS	forms the basic contrasts of a model term (2:4.13.8)
FDESIGNFILE	forms a backing-store file of information for AGDESIGN
FHADAMARDMATRIX	forms Hadamard matrices
FPLOTNUMBER	forms plot numbers for a row-by-column design
FPROJECTIONMATRIX	forms a projection matrix for a set of model terms
XOEFFICIENCY	calculates the efficiency for estimating effects in cross-over designs

7.5 REML analysis of linear mixed models

The REML algorithm estimates the treatment effects and variance components in a linear mixed model: that is, a linear model with both fixed and random effects. Like regression, REML can be used to analyse unbalanced data sets; but, unlike regression, it can account for more than one source of variation in the data, providing an estimate of the variance components associated with the random terms in the model. You can also model the covariance structures on the random terms, such as arise in the analysis of repeated measurements, spatial data and random coefficient regression.

The model for a REML analysis is defined by the commands:

VCOMPONENTS	defines the model for REML (2:5.2.1)
VCYCLE	controls advanced aspects of the REML algorithm (2:5.3.10)
VSTRUCTURE	defines a variance structure for random effects in a REML model (2:5.4.1)
VPEDIGREE	generates an inverse relationship matrix for use in VSTRUCTURE when fitting animal or plant breeding models by REML (2:5.6.1)
VRESIDUAL	defines the residual term for a REML model (2:5.8.2)
VRMETAMODEL	forms the random model for a REML meta analysis (2:5.8.1)
VSTATUS	prints the current model settings for REML (2:5.4.2)

The REML directive carries out the analysis, and a range of other directives and procedures are then available to save results in Genstat data structures, or to produce further information:

REML	fits a variance-component model by residual (or restricted) maximum likelihood (2:5.3.1)
VDISPLAY	displays further output from a REML analysis (2:5.3.2)
VKEEP	copies information from a REML analysis into Genstat data structures (2:5.9.1)
VFRESIDUALS	obtains residuals, fitted values and their standard errors from a REML analysis
VCHECK	checks standardized residuals from a REML analysis (2:5.3.7)
VPREDICT	forms predictions from a REML model (2:5.5.1)
VAIC	calculates the Akaike and Schwarz (Bayesian) information coefficients (2:5.3.8)
VALLSUBSETS	fits all subsets of the fixed terms in a REML analysis
VAYPARALLEL	does the same REML analysis for several y-variates, and collates the output
VBOOTSTRAP	performs a parametric bootstrap of the fixed effects in a REML analysis (2:5.3.6)
VCRITICAL	uses a parametric bootstrap to estimate critical values for

	a fixed term in a REML analysis (2:5.3.6)
VDEFFECTS	plots one- or two-way tables of effects estimated in a REML analysis (2:5.3.4)
VDFIELDRESIDUALS	display residuals from a REML analysis in field layout (2:5.3.5)
VFIXEDTESTS	saves fixed tests from a REML analysis (2:5.9.4)
VFLC	performs an F-test of random effects in a linear mixed model based on linear combinations of the responses, i.e. an FLC test
VFPEDIGREE	checks and prepares pedigree information from several factors, for use by VPEDIGREE and REML (2:5.6.2)
VFUNCTION	calculates functions of variance components from a REML analysis
VGRAPH	plots one- or two-way tables of means from a REML analysis (2:5.3.4)
VHERITABILITY	calculates generalized heritability for a random term in a REML analysis
VMCOMPARISON	performs pairwise comparisons between REML means
VMETA	performs a multi-treatment meta analysis using summary results from individual experiments
VPERMTEST	does random permutation tests for the fixed effects in a REML analysis (2:5.3.6)
VPLOT	plots residuals from a REML analysis (2:5.3.5)
VPOWER	uses a parametric bootstrap to estimate the power (probability of detection) for terms in a REML analysis
VRPERMTEST	performs permutation tests for random terms in REML analysis
VRACCUMULATE	forms a summary accumulating the results of a sequence of REML random models (2:5.3.8)
VRCHECK	checks effects of a random term in a REML analysis (2:5.3.7)
VRFIT	fits terms from a REML fixed model in a Genstat regression
VRADD	adds terms from a REML fixed model into a Genstat regression
VRDISPLAY	displays output for a REML fixed model fitted in a Genstat regression
VRDROP	drops terms in a REML fixed model from a Genstat regression
VRKEEP	saves output for a REML fixed model fitted in a Genstat regression
VRSETUP	sets up Genstat regression to assess terms from a REML fixed model
VRSWITCH	adds or drops terms from a REML fixed model in a Genstat regression
VRTRY	tries the effect of adding and dropping terms from a REML fixed model in a Genstat regression
VSAMPLESIZE	estimates the replication to detect a fixed term or contrast in a REML analysis, using parametric bootstrap
VSCREEN	performs screening tests for fixed terms in a REML analysis (2:5.3.6)
VSOM	analyses a simple REML variance components model for

	outliers using a variance shift outlier model (2:5.3.7)
`VSPREADSHEET`	saves results from a `REML` analysis in a spreadsheet (2:5.9.2)
`VSURFACE`	fits a 2-dimensional spline surface using `REML`, and estimates its extreme point
`VTCOMPARISONS`	calculates comparison contrasts within a multi-way table of predicted means from a REML analysis (2:5.5.2)
`VUVCOVARIANCE`	forms the unit-by-unit variance-covariance matrix for specified variance components in a `REML` model

Procedures are also being developed to to provide automatic selection of `REML` random models for single trials, series of trials and meta analysis.

`VABLOCKDESIGN`	analyses an incomplete-block design by `REML`, allowing automatic selection of random and spatial covariance models
`VAROWCOLUMNDESIGN`	analyses a row-and-column design by `REML`, with automatic selection of the best random and spatial covariance model
`VALINEBYTESTER`	provides combinabilities and deviances for a line-by-tester trial analysed by `VABLOCKDESIGN` or `VAROWCOLUMNDESIGN`
`VLINEBYTESTER`	analyses a line-by-tester trial by `REML`
`VASERIES`	analyses a series of trials with incomplete-block or row-and-column designs by `REML`, automatically selecting the best random models
`VASDISPLAY`	displays further output from an analysis by `VASERIES`
`VASKEEP`	copies information from an analysis by `VASERIES` into Genstat data structures
`VASMEANS`	saves experiment × treatment means from analysis of a series of trials by `VASERIES`
`VAMETA`	performs a `REML` meta analysis of a series of trials
`VFMODEL`	forms a model-definition structure for a `REML` analysis
`VFSTRUCTURE`	adds a covariance-structure definition to a `REML` model-definition structure
`VMODEL`	specifies the model for a `REML` analysis using a model-definition structure defined by `VFMODEL`
`VAOPTIONS`	defines options for the fitting of models by `VARANDOM` and associated procedures
`VARANDOM`	finds the best `REML` random model from a set of models defined by `VFMODEL`
`VARECOVER`	recovers when `REML`, is unable to fit a model, by simplifying the random model

Other useful procedures include:

`FCONTRASTS`	modifies a model formula to contain contrasts of factors
`FDIALLEL`	forms the components of a diallel model for REML or regression
`F2DRESIDUALVARIOGRAM`	calculates and plots a 2-dimensional variogram from a 2-dimensional array of residuals
`TOBIT`	linear mixed model analysis of data with fixed-threshold censoring

7.6 Multivariate and cluster analysis

Many multivariate techniques are implemented as standard Genstat directives. Others are supplied as procedures which make use of the comprehensive toolkit that Genstat provides, for example, matrix calculations, singular value decompositions (4.10.1), and eigenvalue decompositions (4.10.2). References are given in the list, below, for those described in Part 2. Details of the others can be found in the *Genstat Reference Manual*.

`FSSPM`	calculates values for SSPM structures – sums of squares and products, means, etc (2:6.1.1)
`ROBSSPM`	forms robust estimates of sum-of-squares-and-products matrices (2:6.1.1)
`CORRELATE`	forms correlations between variates (2:2.8.1)
`FVCOVARIANCE`	forms the variance-covariance matrix for a list of variates
`FSIMILARITY`	forms a similarity matrix or a between-group similarity matrix from a units-by-variates data matrix (2:6.1.2)
`HREDUCE`	forms a reduced similarity matrix, by groups (2:6.1.3)
`MANTEL`	assesses the association between similarity matrices (2:6.1.5)
`ECANOSIM`	does a nonparametric analysis of similarities (*ANOSIM*) to test for differences between two or more groups of sampling units (2:6.1.6)
`PCP`	principal components analysis (2:6.2.1)
`LRVSCREE`	prints a scree diagram and/or a difference table of latent roots (2:6.2.2)
`CVA`	canonical variates analysis (2:6.3.1)
`CVASCORES`	calculates scores for individual units in canonical variates analysis (2:6.3.2)
`CVAPLOT`	plots mean and unit scores from a canonical variates analysis (2:6.3.3)
`FACROTATE`	rotates factor loadings from a `PCP`, `CVA` or `FCA` (2:6.4)
`DISCRIMINATE`	performs discriminant analysis (2:6.5.1)
`SDISCRIMINATE`	selects the best set of variates to discriminate between groups (2:6.5.2)
`QDISCRIMINATE`	performs quadratic discrimination between groups i.e. allowing for different variance-covariance matrices (2:6.5.3)
`MANOVA`	multivariate analysis of variance and covariance (2:6.6.1)
`RMULTIVARIATE`	multivariate linear regression (2:6.6.2)
`MVAOD`	does an analysis of distance of multivariate data (2:6.6.3)
`RIDGE`	produces ridge regression and principal component regression analyses (2:6.7)
`PLS`	fits a partial least squares regression model (2:6.8)
`OPLS`	performs orthogonal partial least squares regression
`CANCORRELATION`	canonical correlation analysis (2:6.9)
`PCO`	principal coordinates analysis (2:6.10.1)
`ADDPOINTS`	adds points for new objects to a `PCO` (2:6.10.2)
`PCORELATE`	relates principal coordinates to original data variates (2:6.10.3)
`MDS`	non-metric multidimensional scaling (2:6.12)
`CORANALYSIS`	does correspondence analysis, or reciprocal averaging (2:6.13.1)

MCORANALYSIS	does multiple correspondence analysis (2:6.13.2)
CABIPLOT	plots results from correspondence analysis or multiple correspondence analysis (2:6.13.3)
RDA	performs redundancy analysis (2:6.14)
CCA	performs canonical correspondence analysis (2:6.15)
DBIPLOT	plots a biplot from an analysis by PCP, CVA or PCO (2:6.16.1)
CRBIPLOT	plots correlation or distance biplots after RDA, or ranking biplots after CCA (2:6.16.2)
CRTRIPLOT	plots ordination biplots or triplots after RDA or CCA (2:6.16.3)
GGEBIPLOT	plots biplots to assess genotype and genotype-by-environment variation
SKEWSYMMETRY	provides an analysis of skew-symmetry for an asymmetric matrix (2:6.17)
ROTATE	Procrustes rotation (2:6.18.1)
GENPROCRUSTES	generalized Procrustes analysis (2:6.18.2)
PCOPROCRUSTES	performs a multiple Procrustes analysis (2:6.18.3)
HCLUSTER	hierarchical cluster analysis from a similarity matrix (2:6.19.1)
HDISPLAY	displays results associated with hierarchical clustering (2:6.19.2)
HLIST	lists a data matrix in abbreviated form (2:6.19.3)
HSUMMARIZE	summarizes data variates by clusters (2:6.19.4)
DDENDROGRAM	draws dendrograms with control over structure and style (2:6.19.5)
DMST	gives a high resolution plot of an ordination with minimum spanning tree (2:6.19.6)
DCLUSTERLABELS	labels clusters in a single-page dendrogram plotted by DDENDROGRAM (2: 6.19.8)
HCOMPAREGROUPINGS	compares groupings generated, for example, from cluster analyses (2:6.19.7)
HBOOTSTRAP	performs bootstrap analyses to assess the reliability of clusters from hierarchical cluster analysis (2:6.19.8)
HFAMALGAMATIONS	forms an amalgamations matrix from a minimum spanning tree
HFCLUSTERS	forms a set of clusters from an amalgamations matrix (2:6.19.8)
HPCLUSTERS	prints a set of clusters
CLUSTER	non-hierarchical clustering from a data matrix (2:6.20.1)
CLASSIFY	obtains a starting classification for non-hierarchical clustering (2:6.20.2)
BCLASSIFICATION	constructs a classification tree (2:6.21.1)
BCDISPLAY	displays a classification tree (2:6.21.2)
BCKEEP	saves information from a classification tree (2:6.21.5)
BCVALUES	forms values for nodes of a classification tree (2:6.21.3)
BPRUNE	prunes a tree using minimal cost complexity (4.12.8, 2:3.9.3, 2:6.21.3)
BCIDENTIFY	identifies specimens using a classification tree (2:6.21.4)
BCFOREST	constructs a random classification forest
BCFDISPLAY	displays information about a random classification forest

`BCFIDENTIFY`	identifies specimens using a random classification forest
`BKEY`	constructs an identification key (2:6.22.1)
`BKDISPLAY`	displays an identification key (2:6.22.2)
`BKIDENTIFY`	identifies specimens using a key (2:6.22.3)
`BKKEEP`	saves information from an identification key (2:6.22.4)
`IDENTIFY`	identifies an unknown specimen from a defined set of objects (2:6.22.5)
`IRREDUNDANT`	forms irredundant test sets for the efficient identification of a set of objects (2:6.11.6)
`AMMI`	allows exploratory analysis of genotype × environment interactions
`CINTERACTION`	clusters rows and columns of a two-way interaction table
`CONVEXHULL`	finds the points of a single or a full peel of convex-hulls
`DPARALLEL`	displays multivariate data using parallel coordinates (2:2.7.2)
`MULTMISSING`	estimates missing values for units in a multivariate data set
`NORMTEST`	performs tests of univariate and/or multivariate Normality

7.7 Time series analysis

Genstat provides several methods for examining and analysing time series. Sample correlation functions are produced by directive `CORRELATE`:

`CORRELATE`	forms correlations between variates, autocorrelations of variates, and lagged cross-correlations between variates (2:7.1.1)

The analysis of Box-Jenkins models is specified by several directives:

`TSM`	defines Box-Jenkins models (2:7.3.2, 2:7.5.1)
`FTSM`	forms preliminary estimates of parameters in time-series models (2:7.7.1)
`TRANSFERFUNCTION`	specifies input series and transfer-function models for subsequent estimation of a model for an output series (2:7.4.1, 2:7.5.2)
`TFIT`	estimates parameters in Box-Jenkins models for time series (2:7.3.3, 2:7.4.2, 2:7.5.3)

Information can be saved in Genstat data structures, or further output can be produced:

`TDISPLAY`	displays further output after an analysis by `ESTIMATE` (2:7.3.5)
`TKEEP`	saves results after `ESTIMATE` (2:7.3.6, 2:7.5.4)
`TFORECAST`	forecasts future values (2:7.3.7, 2:7.4.3, 2:7.5.5)
`TSUMMARIZE`	displays time series model characteristics (2:7.7.3)

You can filter a time series or perform spectral analysis, using the `TFILTER` and `FOURIER` directives, or perform Kalman filtering with the `KALMAN` procedure.

`TFILTER`	filters time series by time-series models (7.6.1)
`FOURIER`	calculates cosine or Fourier transforms of a real or complex series (7.2.1)
`KALMAN`	calculates estimates from the Kalman filter
`DKALMAN`	plots results from an analysis by `KALMAN`

The Genstat procedure library contains procedures which use the directives described in this chapter, together with graphical presentation of the results, so that standard analyses can be carried out conveniently.

BJESTIMATE	fits an ARIMA model, with forecasts and residual checks (2:7.3.1)
BJFORECAST	plots forecasts of a time series using a previously fitted ARIMA (2:7.3.8)
BJIDENTIFY	displays time series statistics useful for ARIMA model selection (2:7.1.3)
DFOURIER	performs a harmonic analysis of a univariate time series (2:7.2.7)
MCROSSSPECTRUM	performs a spectral analysis of a multiple time series (2:7.2.8)
PERIODTEST	gives periodogram-based tests for white noise in time series
PREWHITEN	filters a time series before spectral analysis
REPPERIODOGRAM	gives periodogram-based analyses for replicated time series
SMOOTHSPECTRUM	forms smoothed spectrum estimates for univariate time series (2:7.2.6)
TVARMA	fits a vector autoregressive moving average (VARMA) model
TVFORECAST	forecasts future values from a vector autoregressive moving average (VARMA) model
TVGRAPH	plots a vector autoregressive moving average (VARMA) model

7.8 Repeated measurements

A repeated-measurements study is one in which subjects (animals, people, plots, etc) are observed on several occasions. Each subject usually receives some randomly allocated treatment, either at the outset or repeatedly through the investigation, and is then observed at successive occasions to see how the treatment effects develop. One way to analyse data sets like this is to use Genstat's REML facilities to model the correlation structure over time (see 7.5).

Alternatively, Genstat has procedures for customized plotting of the observations (or profiles) against time, repeated measures analysis of variance, analyses based on ante-dependence structure or generalized estimating equations, and regression or nonlinear modelling of data where the residuals follow an AR1 or power-distance correlation model.

DREPMEASURES	plots profiles and differences of profiles for repeated measures data (2:8.1.1)
VORTHPOLYNOMIAL	calculates orthogonal polynomial time-contrasts for repeated measures (2:8.1.2)
AREPMEASURES	produces an analysis of variance for repeated measurements (2:8.1.3)
MANOVA	performs multivariate analysis of variance and covariance (2:6.6.1, 2:8.1.4)
RMULTIVARIATE	provides multivariate linear regression with accumulated testing of terms (2:6.6.2)
ANTORDER	assesses order of ante-dependence for repeated measures data (2:8.1.5)
ANTTEST	calculates overall tests based on a specified order of ante-dependence (2:8.1.5)
ANTMVESTIMATE	estimates missing values in repeated measurements using

	ante-dependence structure
RAR1	fits regressions with an AR1 or a power-distance correlation model (2:8.1.6)
NLAR1	fits curves with an AR1 or a power-distance correlation model (2:8.1.6)
CUMDISTRIBUTION	fits frequency distributions to accumulated counts
DTIMEPLOT	produces horizontal bars displaying a continuous time record
GEE	fits models to longitudinal data by generalized estimating equations (2:3.5.10)
VHOMOGENEITY	tests homogeneity of variances
AFCARRYOVER	forms factors to represent carry-over effects in cross-over trials
AGCROSSOVERLATIN	generates Latin squares balanced for carry-over effects (2:4.9.3)

7.9 Survival analysis

In survival data the response variate is the survival time of an individual like a medical patient or an industrial component. The responses are often *censored*, i.e. some individuals survive beyond the end of the study, and so their survival times are unknown. Genstat provides various ways of estimating the *survivor function* (i.e. the probability that an individual is still surviving at each time). You can do nonparametric tests to compare different survival distributions. Finally, you can model the survival times, by assuming that they follow exponential, Weibull or extremevalue distributions, or by fitting a proportional hazards model.

KAPLANMEIER	calculates the Kaplan-Meier estimate of the survivor function (2:8.2.1)
RLIFETABLE	calculates the life-table estimate of the survivor function (2:8.2.3)
RPHFIT	fits the proportional hazards model to survival data as a generalized linear model (2:8.2.5)
RPHCHANGE	modifies a proportional hazards model fitted by RPHFIT (2:8.2.5)
RPHDISPLAY	prints output for a proportional hazards model fitted by RPHFIT (2:8.2.5)
RPHKEEP	saves information from a proportional hazards model fitted by RPHFIT (2:8.2.5)
RPHVECTORS	forms vectors for fitting proportional hazards data as a generalized linear model
RSURVIVAL	models survival times of exponential, Weibull or extreme-value distributions (2:8.2.4)
RSTEST	compares groups of right-censored survival data by nonparametric tests (2:8.2.2)

7.10 Spatial statistics

Spatial data can be analysed by "kriging", a method originating in geostatistics for analysing data distributed in two dimensions. The kriging model specifies how successive measurements of a variable in space are correlated with each other, in terms of a "variogram". This is analogous to the "correlogram" used in the analysis of time series, but for two-dimensional (spatial) data rather than one-dimensional (temporal) data. There are also commands for "cokriging", which models the spatial behaviour of several variables at once (2:8.3.4). This is useful if a variable, that is difficult or expensive to observe, is correlated with other variables that are easier or cheaper.

FVARIOGRAM	forms auto-variograms for individual variates or cross-variograms for pairs of variates (2:8.3.1)
MVARIOGRAM	fits models to an experimental variogram (2:8.3.2)
DVARIOGRAM	plots fitted models to an experimental variogram (2:8.3.3)
KRIGE	calculates kriged estimates using a model fitted to a sample variogram (2:8.3.4)
KCROSSVALIDATION	computes cross-validation statistics for punctual kriging
FCOVARIOGRAM	forms a covariogram structure containing auto-variograms of individual variates and cross-variograms for pairs from a list of variates (2:8.3.6)
MCOVARIOGRAM	fits models to sets of variograms and cross-variograms (2:8.3.7)
DCOVARIOGRAM	plots 2-dimensional auto- and cross-variograms (2:8.3.8)
COKRIGE	calculates kriged estimates using a model fitted to the sample variograms and cross-variograms of a set of variates (2:8.3.9)

There are also procedures for plotting, manipulating and analysing spatial point patterns.

DKSTPLOT	produces diagnostic plots for space-time clustering
DPOLYGON	draws polygons using high-resolution graphics
DPTMAP	draws maps for spatial point patterns using high-resolution graphics
DPTREAD	adds points interactively to a spatial point pattern
DRPOLYGON	reads a polygon interactively from the current graphics device
FHAT	calculates an estimate of the F nearest-neighbour distribution function
FZERO	gives the F function expectation under complete spatial randomness
GHAT	calculates an estimate of the G nearest-neighbour distribution function
GRLABEL	randomly labels two or more spatial point patterns
GRTHIN	randomly thins a spatial point pattern
GRTORSHIFT	performs a random toroidal shift on a spatial point pattern
GRCSR	generates completely spatially random points in a polygon
KCSRENVELOPES	simulates K function bounds under complete spatial randomness
KHAT	calculates an estimate of the K function
KLABENVELOPES	gives bounds for K function differences under random labelling
KSED	calculates s.e. for K function differences under random labelling
KSTHAT	calculates an estimate of the K function in space, time and space-time
KSTMCTEST	performs a Monte-Carlo test for space-time interaction
KSTSE	calculates the standard error for the space-time K function
KTORENVELOPES	gives bounds for the bivariate K function under independence

`K12HAT`	calculates an estimate of the bivariate K function
`MSEKERNEL2D`	estimates the mean square error for a kernel smoothing
`PTAREAPOLYGON`	calculates the area of a polygon
`PTBOX`	generates a box bounding or surrounding a spatial point pattern
`PTCLOSEPOLYGON`	closes open polygons
`PTDESCRIBE`	gives summary and second order statistics for a point process
`PTGRID`	generates a grid of points in a polygon
`PTINTENSITY`	calculates the overall density for a spatial point pattern
`PTKERNEL2D`	performs kernel smoothing of a spatial point pattern
`PTK3D`	performs kernel smoothing of space-time data
`PTREMOVE`	removes points interactively from a spatial point pattern
`PTROTATE`	rotates a point pattern
`PTSINPOLYGON`	returns points inside or outside a polygon

7.11 Six sigma

Genstat has wide range of facilities to support the six-sigma approach to quality improvement. It can display many different types of control chart.

`SPCCHART`	plots c or u charts representing numbers of defective items (2:2.10.5)
`SPCUSUM`	prints CUSUM tables for controlling a process mean (2:2.10.2)
`SPEWMA`	plots exponentially weighted moving-average control charts (2:2.10.3)
`SPPCHART`	plots p or np charts for binomial testing for defective items (2:2.10.4)
`SPSHEWHART`	plots control charts for mean and standard deviation or range (2:2.10.1)

It can test for Normality, display Pareto charts and calculate capability statistics.

`NORMTEST`	performs tests of univariate and/or multivariate Normality
`SPCAPABILITY`	calculates capability statistics (2:2.10.6)
`TABSORT`	sorts tables to put margins are in ascending or descending order for display as a Pareto chart (4.11.6)

And, of course, it also provides full statistical backup for wider-ranging investigations.

7.12 Survey analysis

There are several procedures for analysing the results of stratified surveys. For details see Part 3 of the *Genstat Reference Manual*.

`SVBOOT`	bootstraps data from random surveys
`SVCALIBRATE`	performs generalized calibration of survey data
`SVGLM`	fits generalized linear models to survey data
`SVHOTDECK`	performs hot-deck and model-based imputation for survey data
`SVREWEIGHT`	modifies survey weights adjusting to ensure that their overall sum weights remains unchanged
`SVSAMPLE`	constructs stratified random samples
`SVSTRATIFIED`	analyses stratified random surveys by expansion or ratio

	raising
`SVTABULATE`	tabulates data from random surveys, including multistage surveys and surveys with unequal probabilities of selection
`SVWEIGHT`	forms survey weights
`CSPRO`	reads a data set from a CSPro survey data file and dictionary, loads it into Genstat or puts it into a spreadsheet file

7.13 Ecological data

Procedures are available to study ecological issues, such as species diversity and abundance.

`ECDIVERSITY`	calculates measures of diversity with jackknife or bootstrap estimates (2:2.11.1)
`ECABUNDANCEPLOT`	produces rank/abundance, *ABC* and *k*-dominance plots (2:2.11.2)
`ECFIT`	fits models to species abundance data (2.11.3)
`ECNICHE`	generates relative abundance of species for niche-based models (2:2.11.4)
`ECRAREFACTION`	calculates individual or sample-based rarefaction (2:2.11.5)
`ECACCUMULATION`	plots species accumulation curves for samples or individuals (2:2.11.6)
`ECNPESTIMATE`	calculates nonparametric estimates of species richness (2:2.11.7)
`ECANOSIM`	does a nonparametric analysis of similarities (*ANOSIM*) to test for differences between two or more groups of sampling units (2:6.1.6)
`LORENZ`	plots the Lorenz curve and calculates the Gini and asymmetry coefficients (2:2.11.8)

7.14 Statistical genetics and QTL estimation

Genstat has a suite of procedures for statistical genetics. Several of these use `REML` to estimate QTLs from single environment, multi-environment and mult-trait trials.

`DQMAP`	displays a genetic map
`DQMKSCORES`	plots a grid of marker scores for genotypes and indicates missing data
`DQMQTLSCAN`	plots the results of a genome-wide scan for QTL effects in multi-environment trials
`DQRECOMBINATIONS`	plots a matrix of recombination frequencies between markers
`DQSQTLSCAN`	plots the results of a genome-wide scan for QTL effects in single-environment trials
`GPREDICTION`	produces genomic predictions (breeding values) using phenotypic and molecular marker information
`QBESTGENOTYPES`	sorts individuals of a segregating population by their genetic similarity with a defined target genotype, using the identity by descent (IBD) information at QTL positions for one or more traits
`QCANDIDATES`	selects QTLs on the basis of a test statistic profile along the genome

`QDESCRIBE`	prints summary statistics of genotypes
`QEIGENANALYSIS`	uses principal components analysis and the Tracy-Widom statistic to find the number of significant principal components to represent a set of variables
`QEXPORT`	exports genotypic data for QTL analysis
`QFLAPJACK`	creates a Flapjack project file from genotypic and phenotypic data
`QGSELECT`	obtains a representative selection of genotypes by means of genetic distance sampling or genetic distance optimization
`QIBDPROBABILITIES`	reads molecular marker data and calculates IBD probabilities
`QIMPORT`	imports genotypic and phenotypic data for QTL analysis
`QKINSHIPMATRIX`	forms a kinship matrix from molecular markers
`QLDDECAY`	estimates linkage disequilibrium (LD) decay along a chromosome
`QLINKAGEGROUPS`	forms linkage groups using marker data from experimental populations
`QMAP`	constructs genetic linkage maps using marker data from experimental populations
`QMASSOCIATION`	performs multi-environment marker trait association analysis in a genetically diverse population using bi-allelic and multi-allelic markers
`QMATCH`	matches different data structures to be used in QTL estimation
`QMBACKSELECT`	performs a QTL backward selection for loci in multi-environment trials or multiple populations
`QMKDIAGNOSTICS`	generates descriptive statistics and diagnostic plots of molecular marker data
`QMESTIMATE`	calculates QTL effects in multi-environment trials or multiple populations
`QMKRECODE`	recodes marker scores into separate alleles
`QMKSELECT`	obtains a representative selection of markers by means of genetic distance sampling or genetic distance optimization
`QMQTLSCAN`	performs a genome-wide scan for QTL effects (Simple and Composite Interval Mapping) in multi-environment trials or multiple populations
`QMTBACKSELECT`	performs a QTL backward selection for loci in multi-trait trials
`QMTESTIMATE`	calculates QTL effects in multi-trait trials
`QMTQTLSCAN`	performs a genome-wide scan for QTL effects (Simple and Composite Interval Mapping) in multi-trait trials
`QMVAF`	calculates percentage variance accounted for by QTL effects in a multi-environment analysis
`QMVESTIMATE`	replaces missing molecular marker scores using conditional genotypic probabilities
`QMVREPLACE`	replaces missing marker scores with the mode scores of the most similar genotypes
`QRECOMBINATIONS`	calculates the expected numbers of recombinations and the recombination frequencies between markers
`QREPORT`	creates an HTML report from QTL linkage or association

	analysis results
QSASSOCIATION	performs marker trait association analysis in a genetically diverse population using bi-allelic and multi-allelic markers
QSBACKSELECT	performs a QTL backward selection for loci in single-environment trials
QSELECTIONINDEX	calculates (molecular) selection indexes by using phenotypic information and/or molecular scores of multiple traits
QSESTIMATE	calculates QTL effects in single-environment trials
QSIMULATE	simulates marker data and QTL effects for single and multiple environment trials
QSQTLSCAN	performs a genome-wide scan for QTL effects (Simple and Composite Interval Mapping) in single-environment trials
QTHRESHOLD	calculates a threshold to identify a significant QTL
VGESELECT	selects the best variance-covariance model for a set of environments

7.15 Microarray data

There is a suite of procedures for the design, analysis and visualization of two-colour and Affymetrix microarray data. These are used by the Microarray menus in Genstat *for Windows*.

AGBIB	generates balanced incomplete block designs
AGLOOP	generates loop designs e.g. for time-course microarray experiments
AGREFERENCE	generates reference-level designs e.g. for microarray experiments
MADESIGN	assesses the efficiency of a two-colour microarray design
MACALCULATE	corrects and transforms two-colour microarray differential expressions
MNORMALIZE	normalizes two-colour microarray data
MAESTIMATE	estimates treatment effects from a two-colour microarray design
AFFYMETRIX	estimates expression values for Affymetrix slides.
MABGCORRECT	performs background correction of Affymetrix slides
MAROBUSTMEANS	does a robust means analysis for Affymetrix slides
MARMA	calculates Affymetrix expression values
MAANOVA	does analysis of variance for a single-channel microarray design
MAREGRESSION	does regressions for single-channel microarray data
MAVDIFFERENCE	applies the average difference algorithm to Affymetrix data
DMADENSITY	plots the empirical CDF or PDF (kernel smoothed) by groups
MAHISTOGRAM	plots histograms of microarray data
MAPLOT	produces two-dimensional plots of microarray data
MASHADE	produces shade plots to display spatial variation of microarray data
MAVOLCANO	produces volcano plots of microarray data
MAPCLUSTER	clusters probes or genes with microarray data
MASCLUSTER	clusters microarray slides

`MA2CLUSTER`	performs a two-way clustering of microarray data by probes (or genes) and slides
`FDRBONFERRONI`	estimates false discovery rates by a Bonferroni-type procedure
`FDRMIXTURE`	estimates false discovery rates using mixture distributions
`MAEBAYES`	modifies t-values by an empirical Bayes method.
`MPOLISH`	performs a median polish of two-way data
`QNORMALIZE`	performs quantile normalization
`THINPLATE`	calculates the basis functions for thin-plate splines
`TUKEYBIWEIGHT`	estimates means using the Tukey biweight algorithm

7.16 Data mining

Genstat provides many conventional statistical techniques that are useful for data mining, including regression (2:3.1, 2:3.2, 2:3.3), log-linear models (2:3.5.1), generalized additive models (2:3.5.7), discriminant analysis (2:6.5) and cluster analysis (2:6.19, 2:6.20). It also provides various more specialized techniques such as association rules, classification and regression trees, random forests, *k*-nearest-neighbours classification, self-organizing maps, neural networks and radial basis functions.

`ASRULES`	derives association rules from transaction data
`BCLASSIFICATION`	constructs a classification tree (2:6.21.1)
`BCDISPLAY`	displays a classification tree (2:6.21.2)
`BCIDENTIFY`	identifies specimens using a classification tree (2:6.21.4)
`BCVALUES`	forms values for nodes of a classification tree (2:6.21.3)
`BCFOREST`	constructs a random classification forest
`BCFDISPLAY`	displays information about a random classification forest
`BCFIDENTIFY`	identifies specimens using a random classification forest
`BREGRESSION`	constructs a regression tree (2:3.9.1)
`BRDISPLAY`	displays a regression tree (2:3.9.2)
`BRPREDICT`	makes predictions using a regression tree (2:3.9.4)
`BRVALUES`	forms values for nodes of a regression tree (2:3.9.3)
`KNEARESTNEIGHBOURS`	classifies items or predicts their responses by examining their *k* nearest neighbours
`NNFIT`	fits a multi-layer perceptron neural network
`NNDISPLAY`	displays output from a multi-layer perceptron neural network fitted by `NNFIT`
`NNPREDICT`	forms predictions from a multi-layer perceptron neural network fitted by `NNFIT`
`RBFIT`	fits a radial basis function model
`RBDISPLAY`	displays output from a radial basis function model fitted by `RBFIT`
`RBPREDICT`	forms predictions from a radial basis function model fitted by `RBFIT`
`SOM`	declares a self-organizing map
`SOMADJUST`	performs adjustments to the weights of a self-organizing map
`SOMDESCRIBE`	summarizes values of variables at nodes of a self-organizing map
`SOMESTIMATE`	estimates the weights for self-organizing maps
`SOMIDENTIFY`	allocates samples to nodes of a self-organizing map
`SOMPREDICT`	makes predictions using a self-organizing map

`SVMFIT`	fits a support vector machine
`SVMPREDICT`	forms the predictions using a support vector machine

7.17 Other statistical methods

Genstat provides procedures for several other statistical methods, including Jackknife, Bootstrap and Bayesian analyses, that are not described in the *Guide*. Details can be found in Part 3 of the *Genstat Reference Manual*.

References

Chapter 4

Bowdler, H., Martin, R.S., Reinsch, C. & Wilkinson, J.H. (1968). The QR and QL algorithms for symmetric matrices. *Numerische Mathematik* **11**, 293-306.

Breiman, L., Friedman, J.H., Olshen, R.A. & Stone, C.J. (1984). *Classification and Regression Trees*. Wadsworth, Monterey.

Digby, P.G.N. & Kempton, R.A. (1987). *Multivariate Analysis of Ecological Communities*. Chapman & Hall, London.

Eckart, C. & Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika* **1**, 211-218.

Gehan, E.A. (1965). A generalized Wilcoxon test for comparing arbitrarily singly-censored samples. *Biometrika*, **52**, 203-223

Golub, G.H. & Reinsch, C. (1971). Singular value decomposition and least squares solutions. *Numerische Mathematik* **14**, 403-420.

Herraman, C. (1968). Algorithm AS12: Sums of squares and products matrix. *Applied Statistics* **17**, 289-292.

Martin, R.S., Reinsch, C. & Wilkinson, J.H. (1968). Householders tridiagonalisation of a symmetric matrix. *Numerische Mathematik* **11**, 181-195.

Rao, C.R. (1973). *Linear statistical inference and its applications*. Wiley, New York.

Taylor, P.C. & Silverman, B.W. (1993). Block diagrams and splitting criteria for classification trees. *Statistics & Computing*, **3**, 147-161.

Wichmann, B.A. & Hill, I.D. (1982). An efficient and portable pseudo-random number generator. *Applied Statistics* **31**, 188-190.

Index