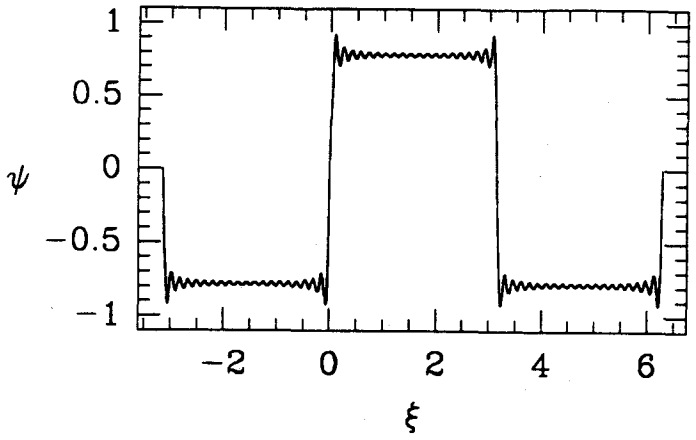
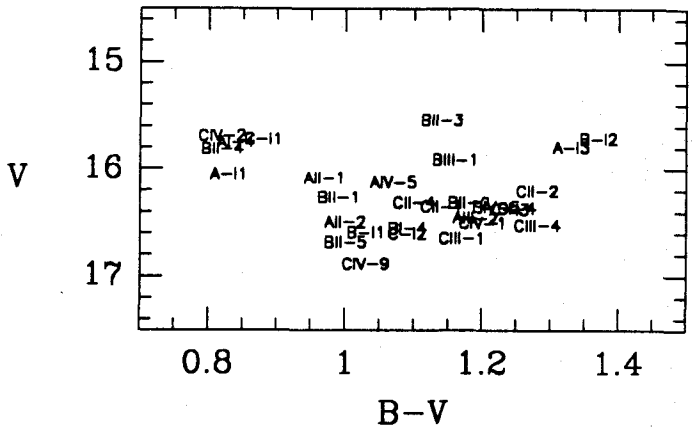
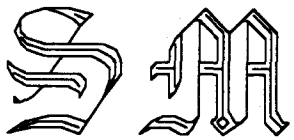
New Folder Name Program Handbook

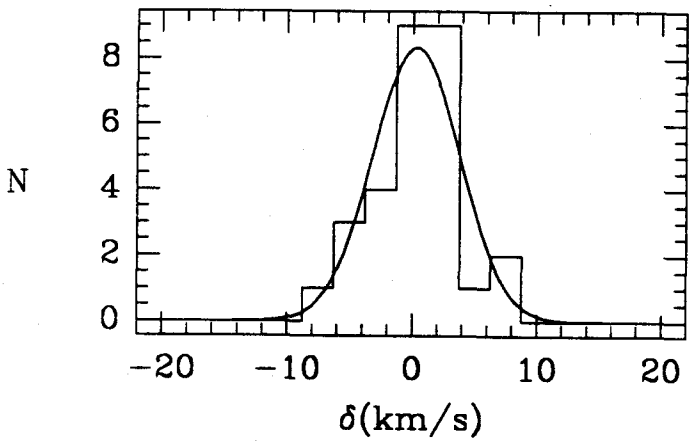


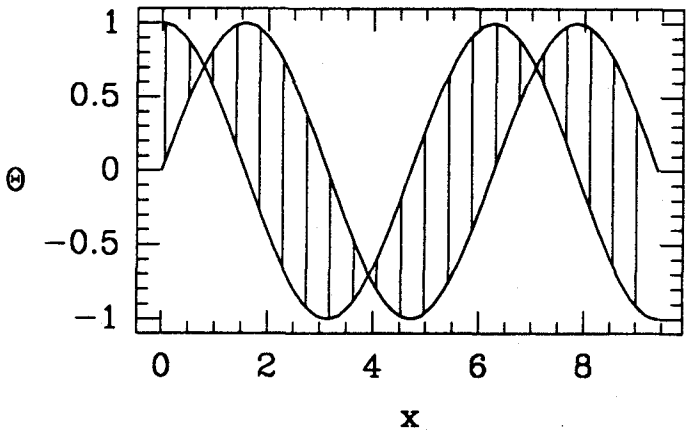




Robert Lupton
University of Hawaii

Patricia Monger
McMaster University
University of Toronto

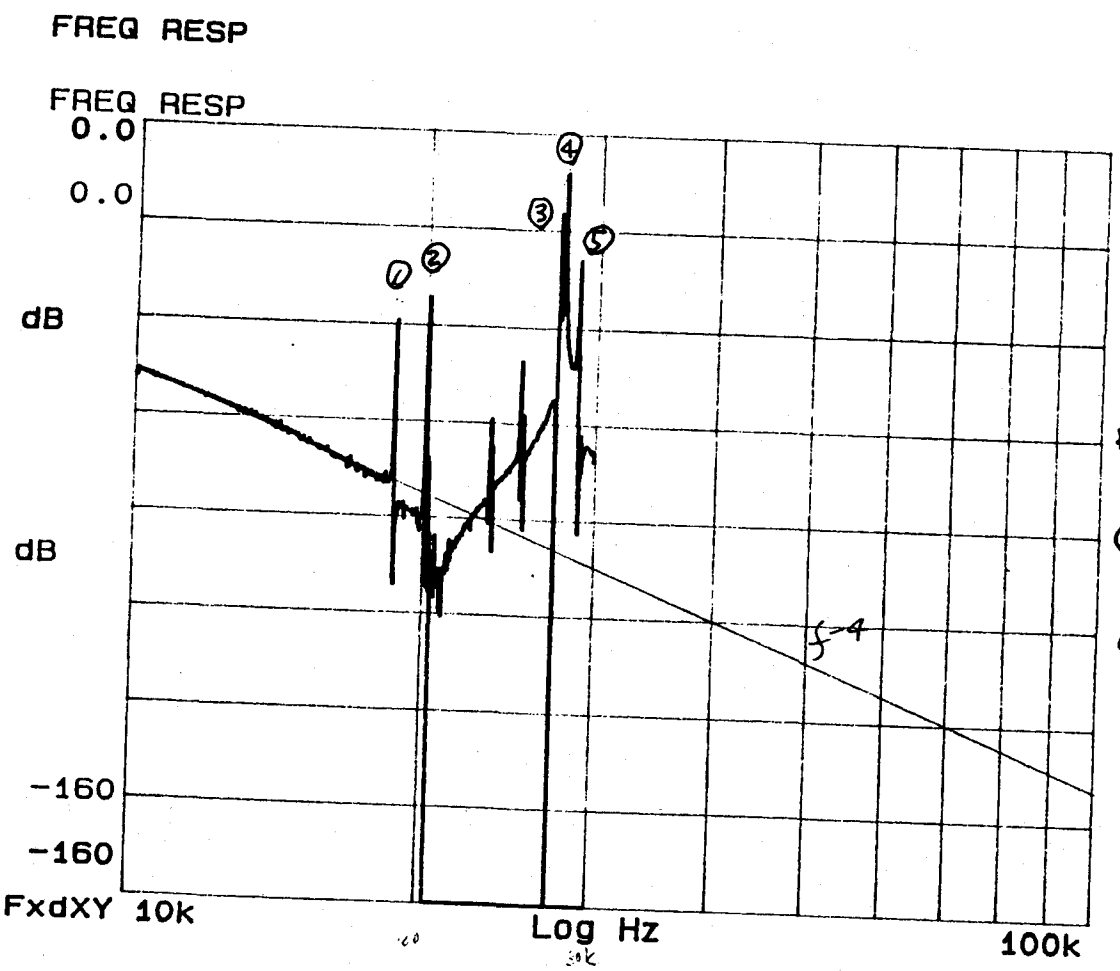


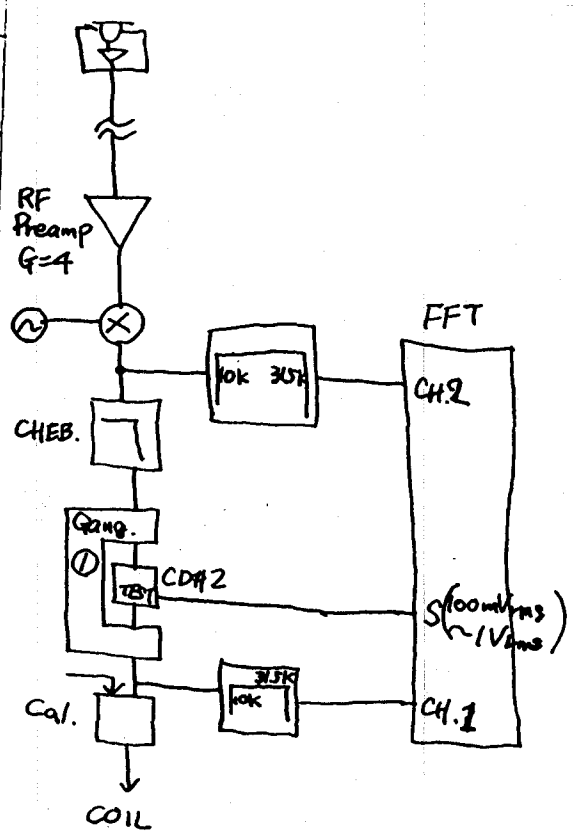


Sugi Measurements

Oct. 16, 91
16:53

XF from A2 coil vol. to Demodulated signal





10k ~ 20k : "SM911016"

20k ~ 30k : "SN911016"

in 10/14/91 mtg

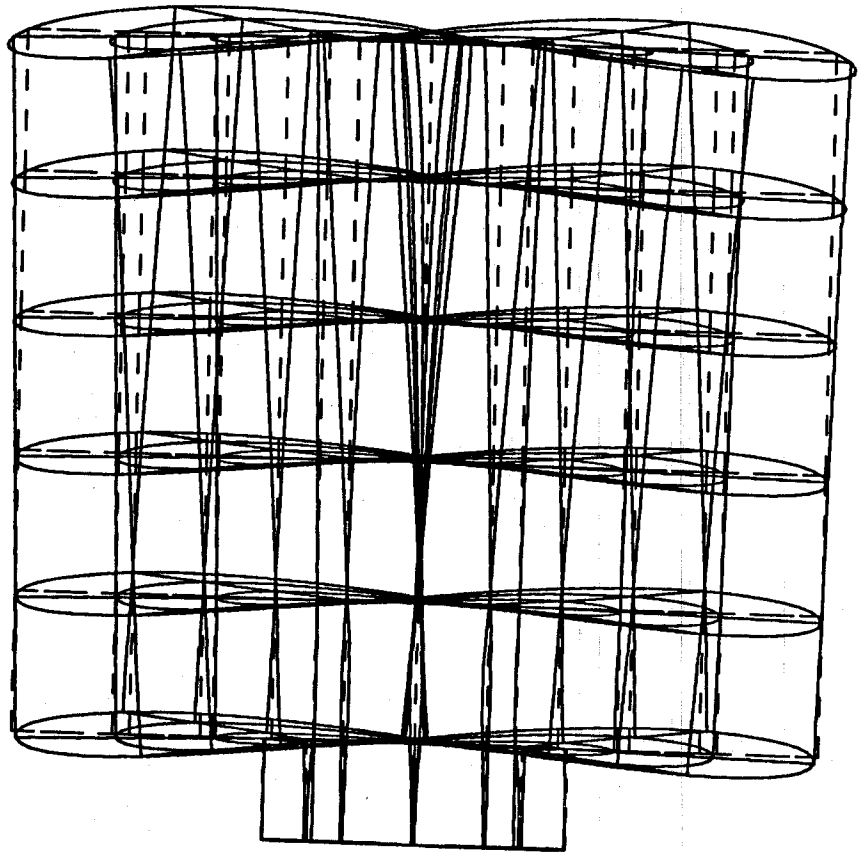
① 18.675 kHz
② 20.15 kHz
③ 27.375 kHz
④ 27.70 kHz
⑤ 28.69 kHz

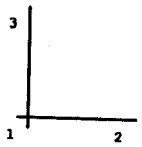
	Around peak	Other point
Freq. Resolution	1.25 Hz	6 ~ 50 Hz
Integratio Time	3 sec	200 msec ~ 1 sec

$\lambda = 18.977$ KHz

ABAQUS

U
MAG. FACTOR = +4.9E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

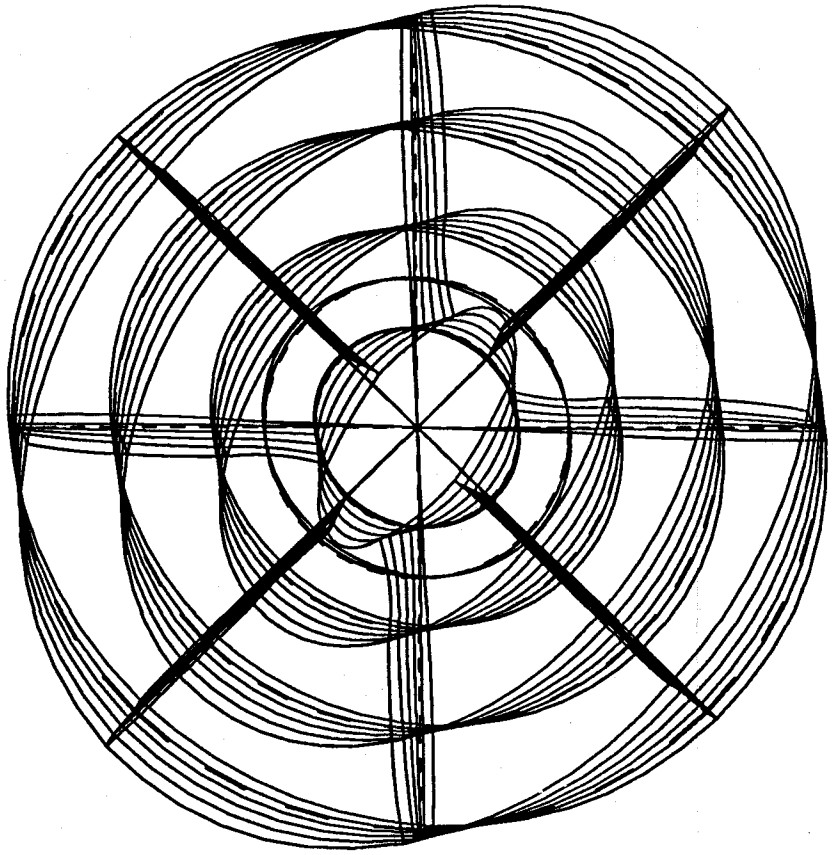
EIGENVALUE = +1.421E+10

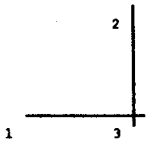
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 18.977$ KHz

ABAQUS

U
MAG. FACTOR = +4.8E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

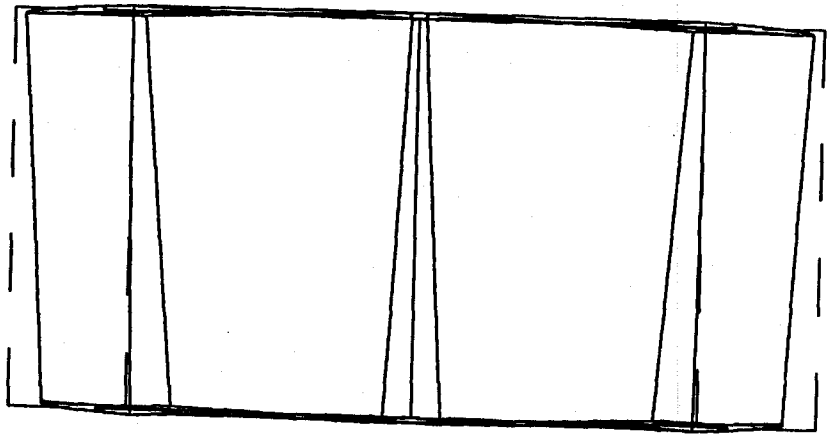
EIGENVALUE = +1.421E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 18.977$ KHz

ABAQUS

U
MAG. FACTOR = +1.4E+00
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1

Mode analysis of test mass

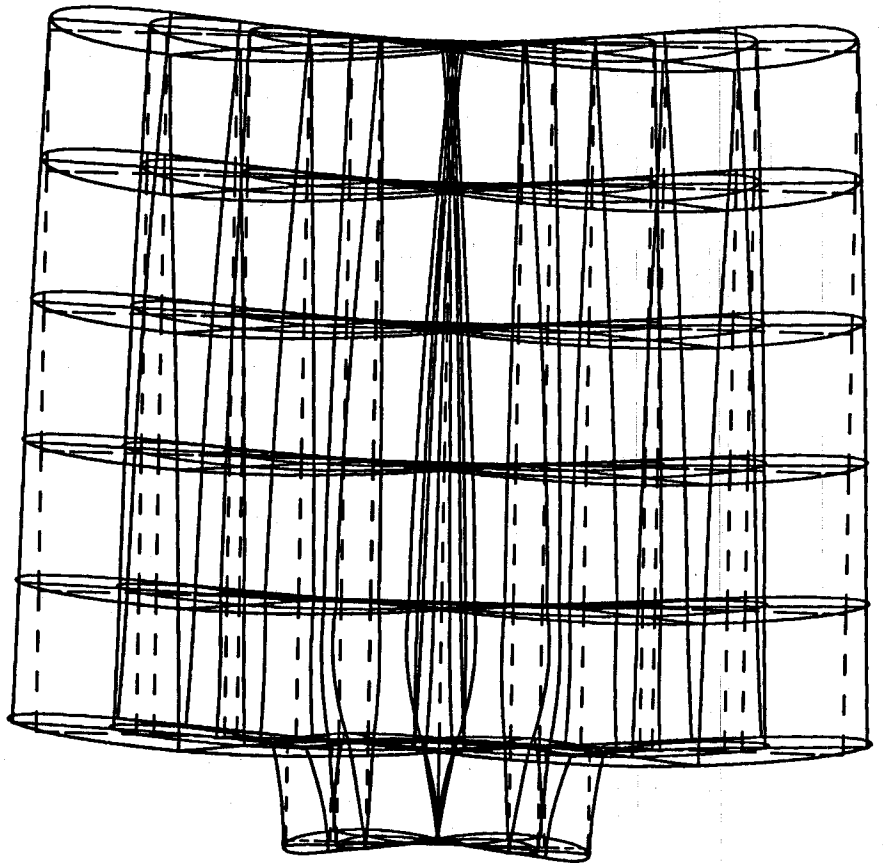
EIGENVALUE = +1.421E+10

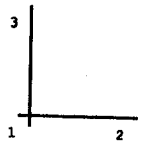
3 ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 20.817\ KHz$

ABAQUS

U
MAG. FACTOR = +5.2E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

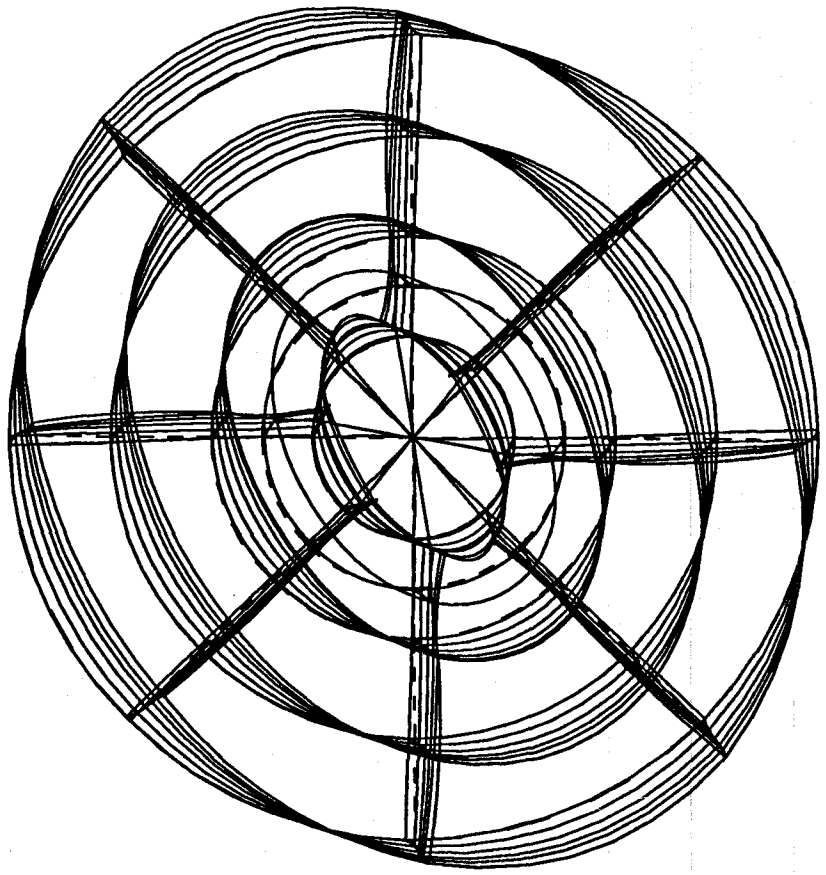
EIGENVALUE = +1.710E+10

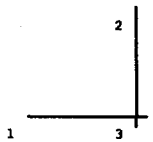
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 20.817$ KHz

ABAQUS

U
MAG. FACTOR = +4.9E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

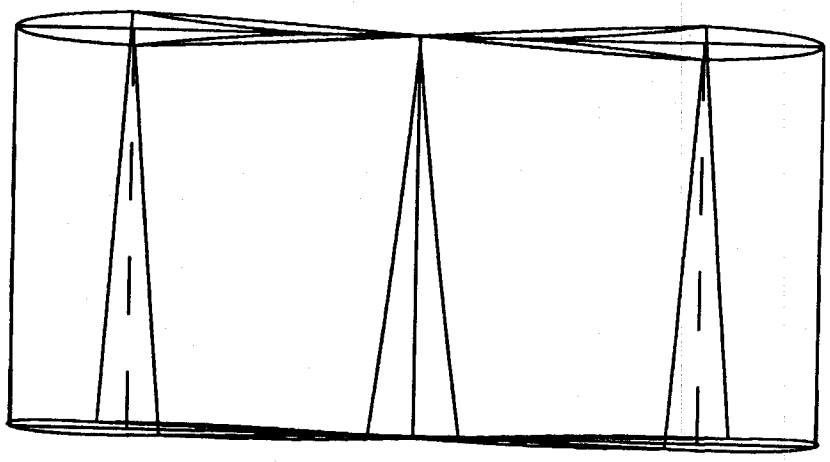
EIGENVALUE = +1.710E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 20.817$ KHz

ABAQUS

U
MAG. FACTOR = +2.6E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1 3

Mode analysis of test mass

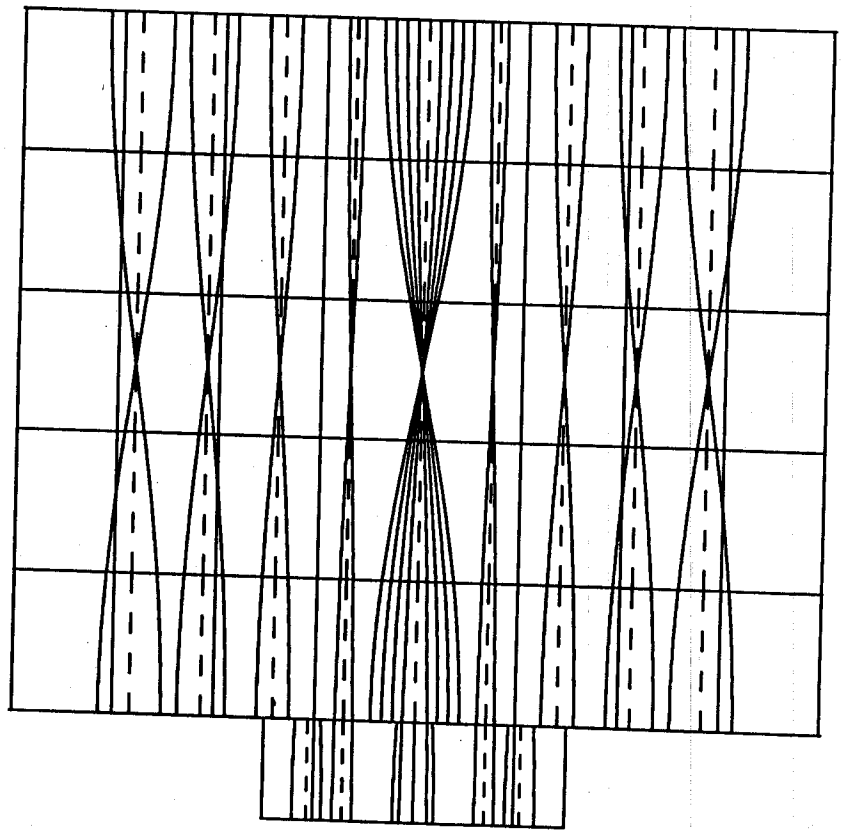
EIGENVALUE = +1.710E+10

ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 21.006$ KHz

ABAQUS

U
MAG. FACTOR = +5.6E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



Mode analysis of test mass

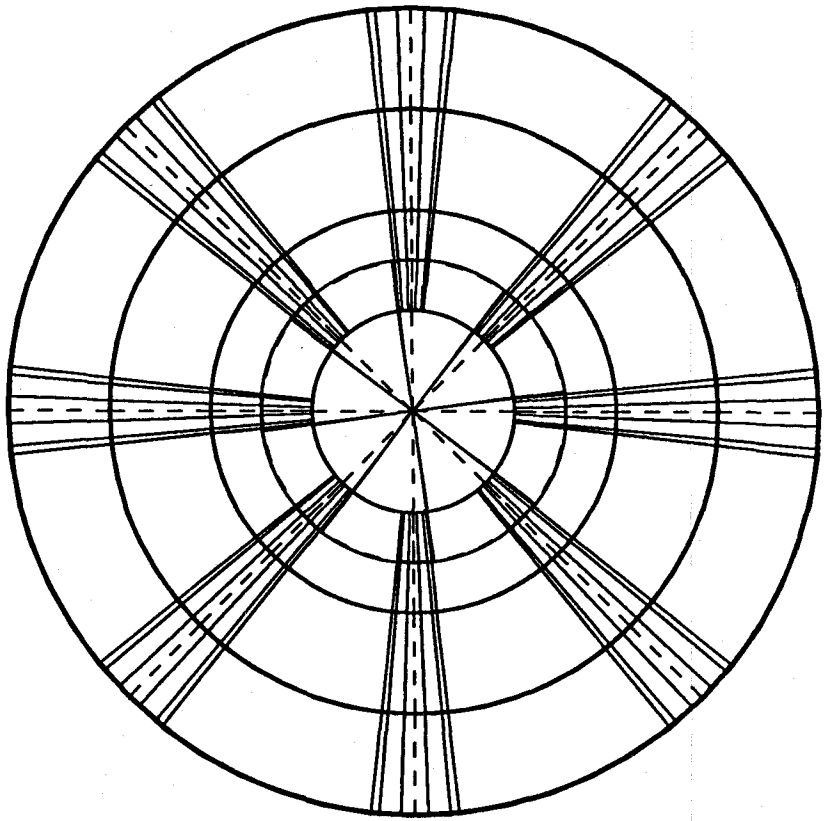
EIGENVALUE = +1.741E+10

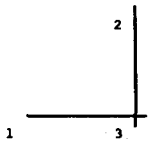
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 21.006$ kHz

ABAQUS

U
MAG. FACTOR = +5.6E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

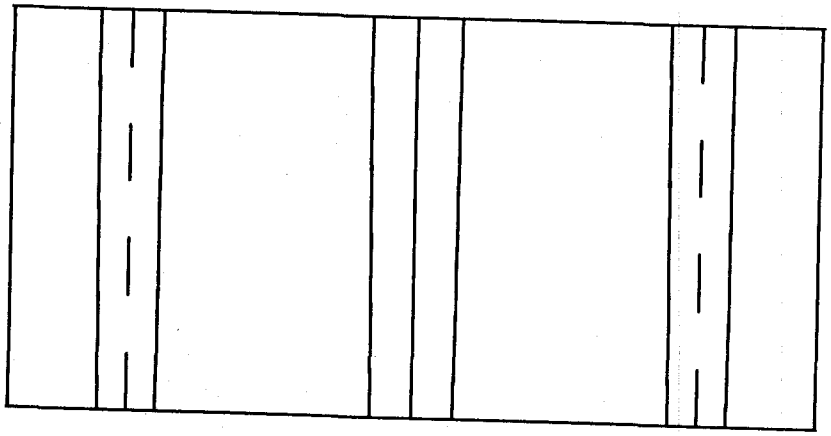
EIGENVALUE = +1.741E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 21.006$ KHz

ABAQUS

U
MAG. FACTOR = +4.5E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1 3

Mode analysis of test mass

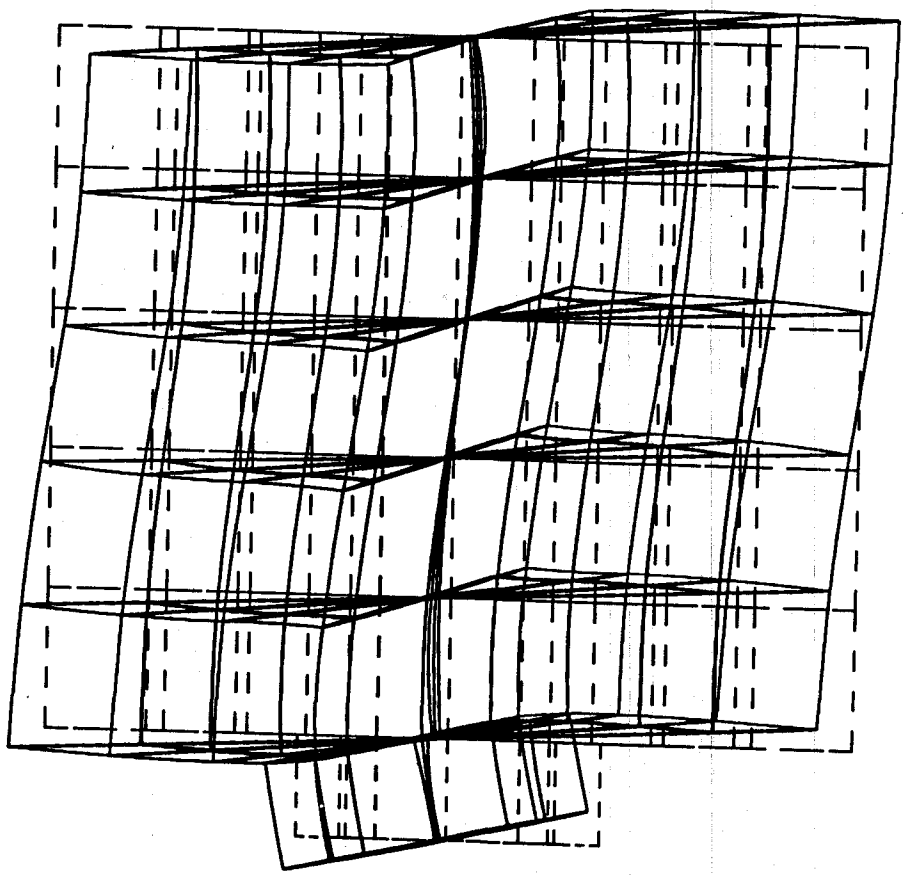
EIGENVALUE = +1.741E+10

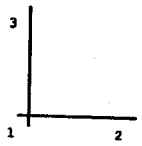
ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 23.636$ kHz

ABAQUS

U
MAG. FACTOR = +4.5E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

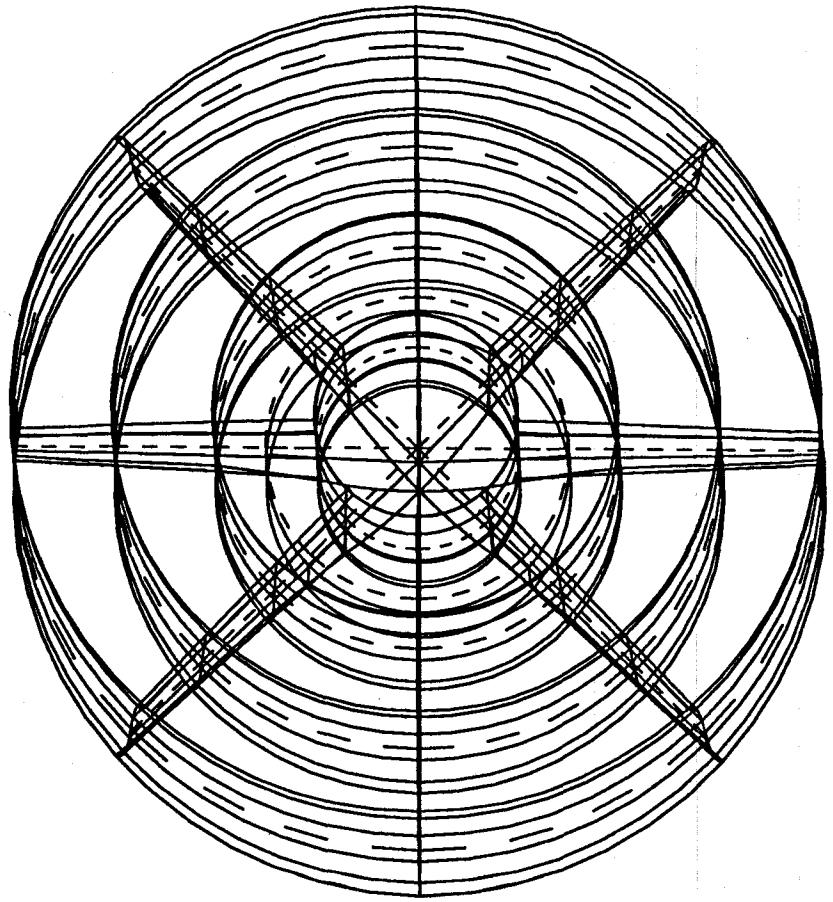
EIGENVALUE = +2.205E+10

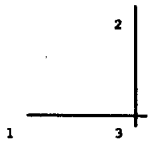
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 23.636$ KHz

ABAQUS

U
MAG. FACTOR = +5.7E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

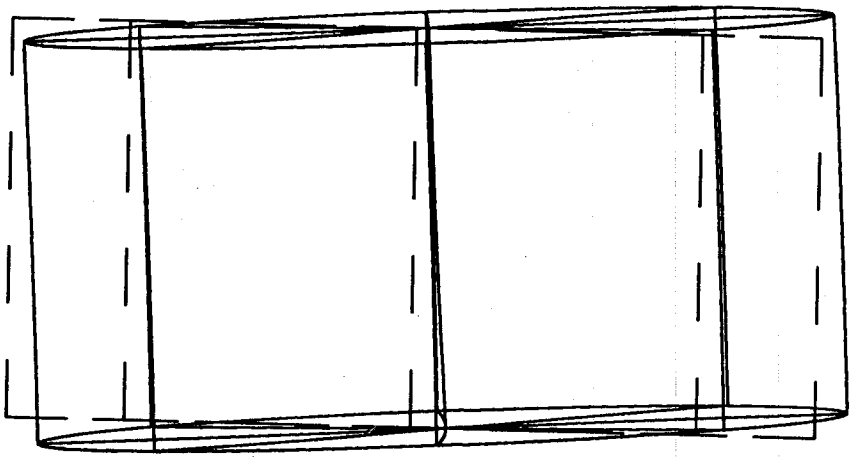
EIGENVALUE = +2.205E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 23.636$ KHz

ABAQUS

U
MAG. FACTOR = +1.3E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1

Mode analysis of test mass

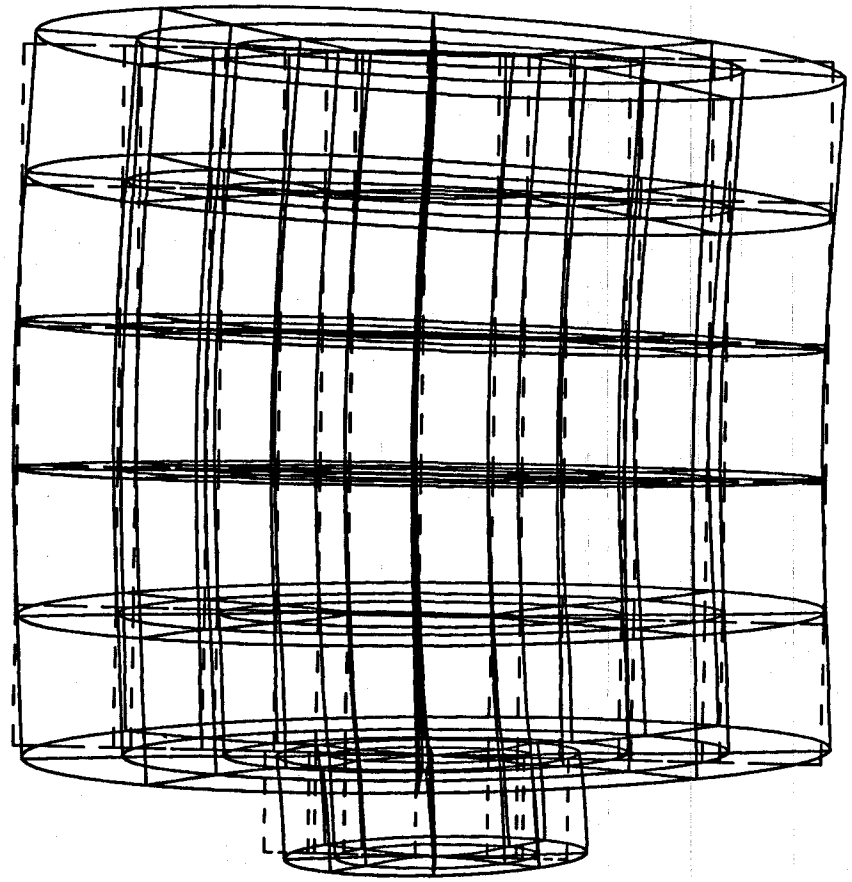
EIGENVALUE = +2.205E+10

3 ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 25.292$ kHz

ABAQUS

U
MAG. FACTOR = +7.2E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



Mode analysis of test mass

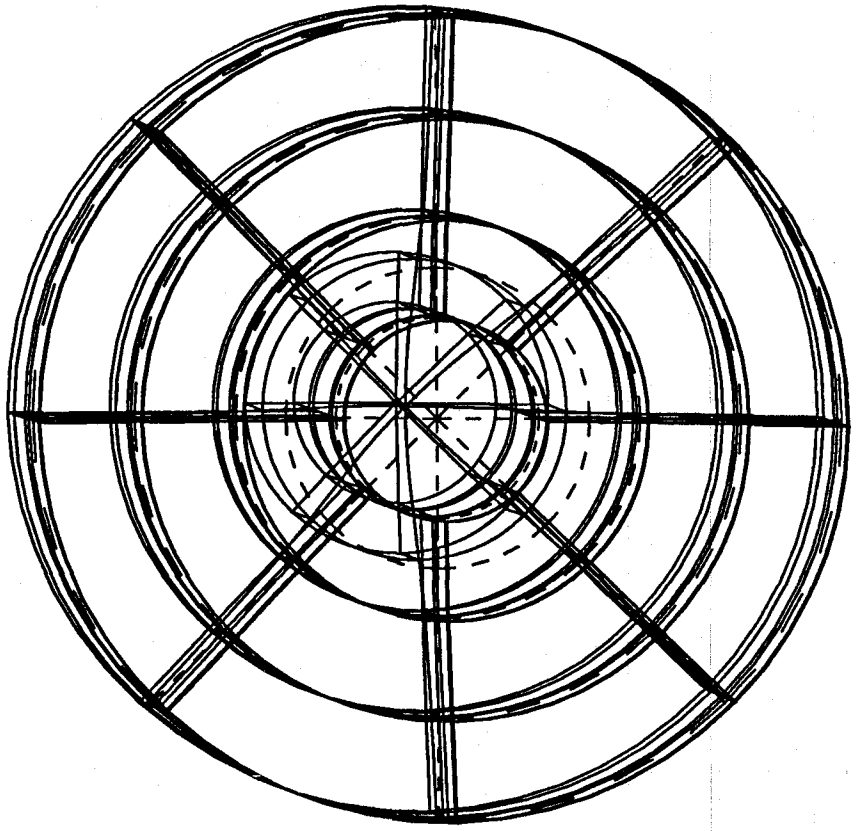
EIGENVALUE = +2.525E+10

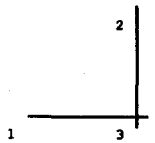
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 25.292$ kHz

ABAQUS

U
MAG. FACTOR = +5.3E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

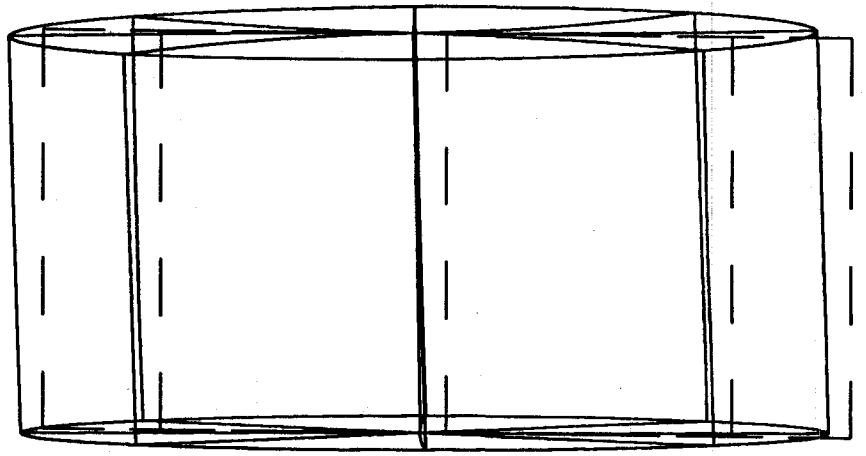
EIGENVALUE = +2.525E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 11:42:33

$\lambda = 25.292$ KHz

ABAQUS

U
MAG. FACTOR = +3.9E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1
3

Mode analysis of test mass

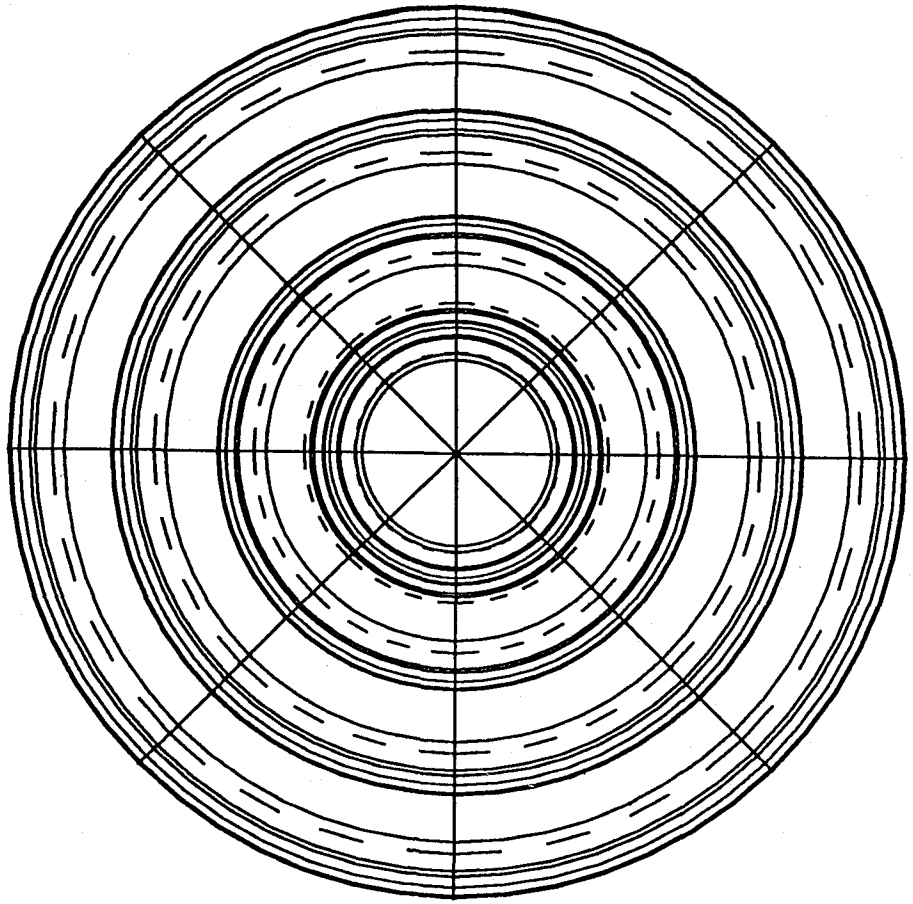
EIGENVALUE = +2.525E+10

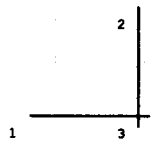
ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 27.520$ KHz

ABAQUS

U
MAG. FACTOR = +1.4E+00
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

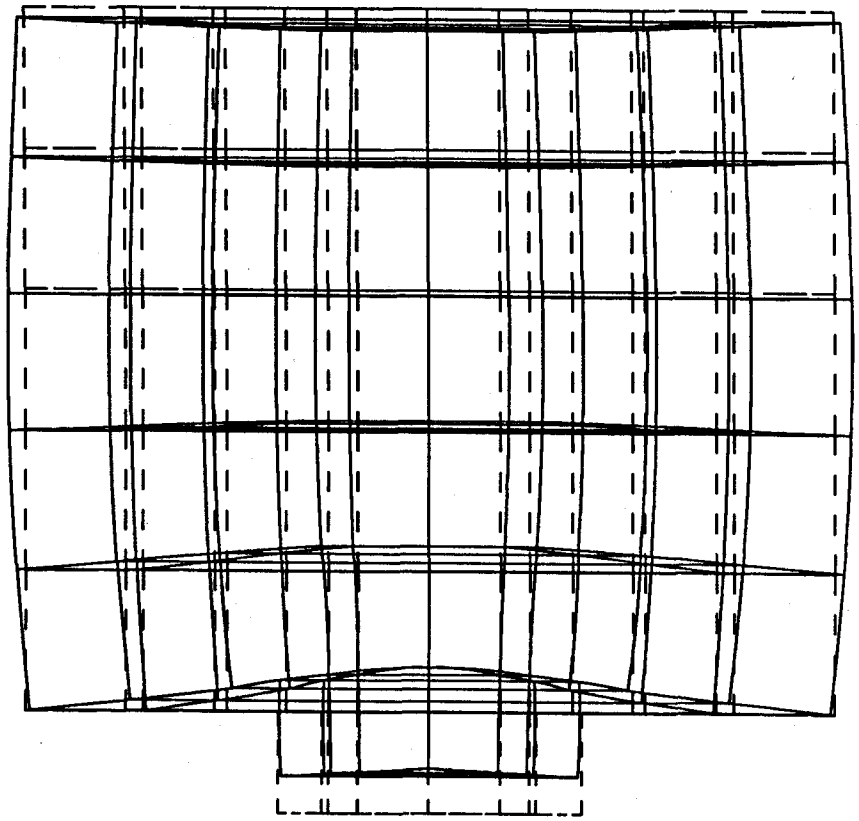
EIGENVALUE = +2.989E+10

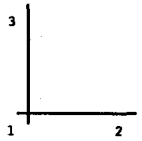
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 27.520$ KHz

ABAQUS

U
MAG. FACTOR = +5.6E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

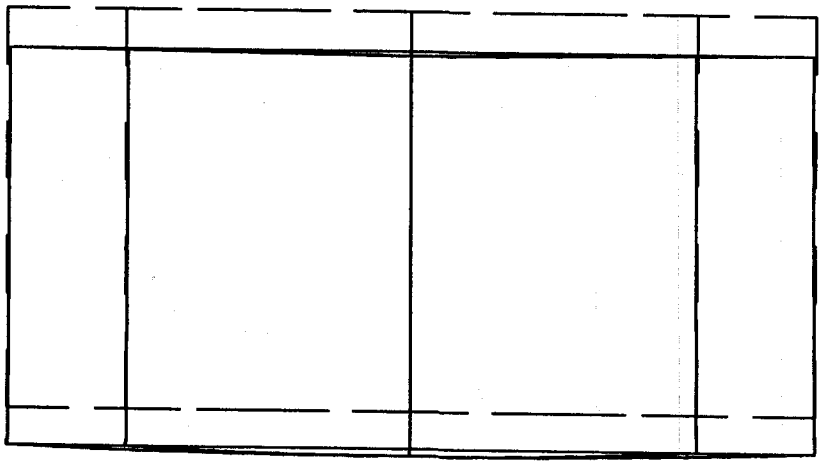
EIGENVALUE = +2.989E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 27.520$ kHz

ABAQUS

U
MAG. FACTOR = +1.4E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1 3

Mode analysis of test mass

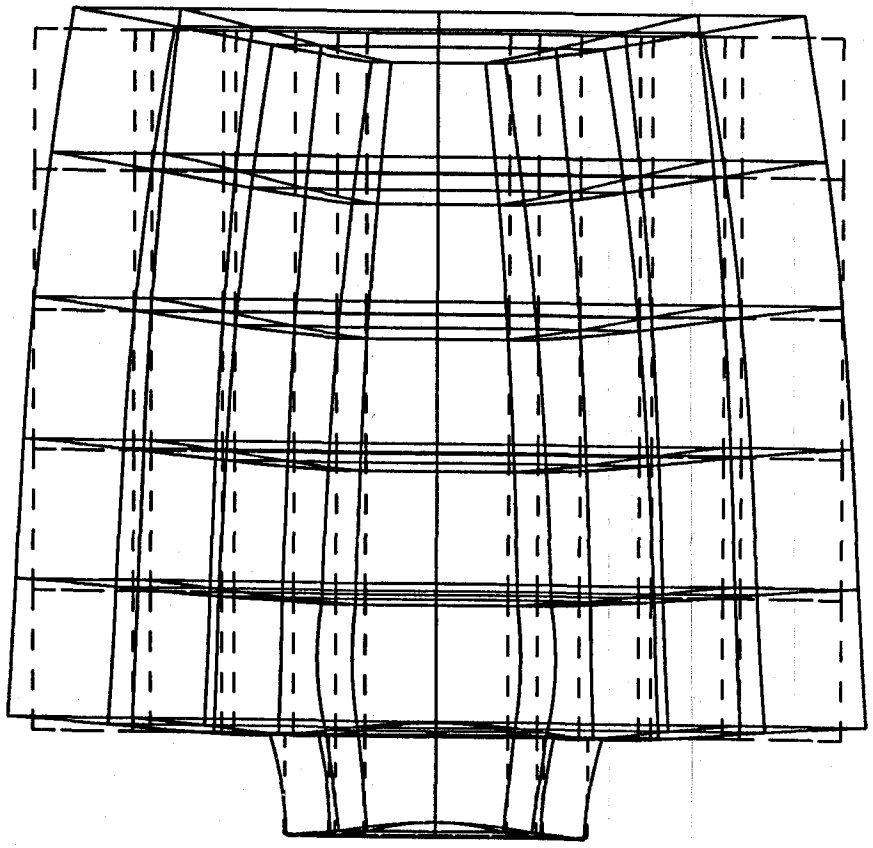
EIGENVALUE = +2.989E+10

ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 28.187$ KHz

ABAQUS

U
MAG. FACTOR = +4.9E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



Mode analysis of test mass

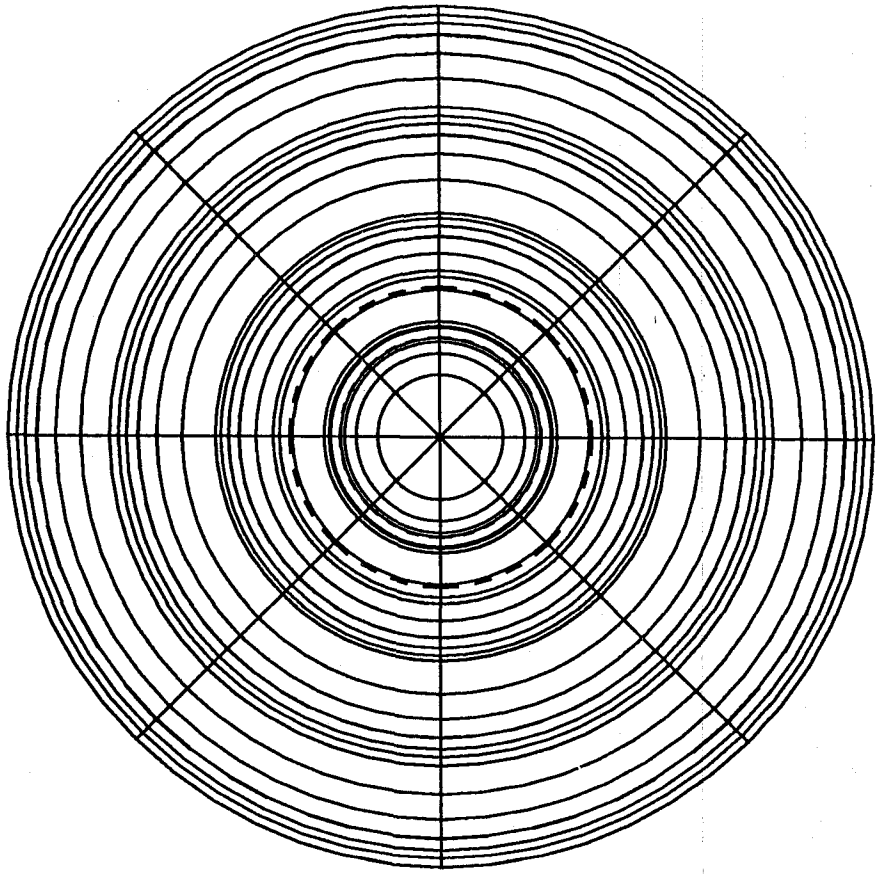
EIGENVALUE = +3.136E+10

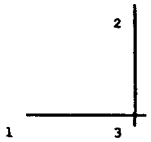
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 28.187$ kHz

ABAQUS

U
MAG. FACTOR = +5.6E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

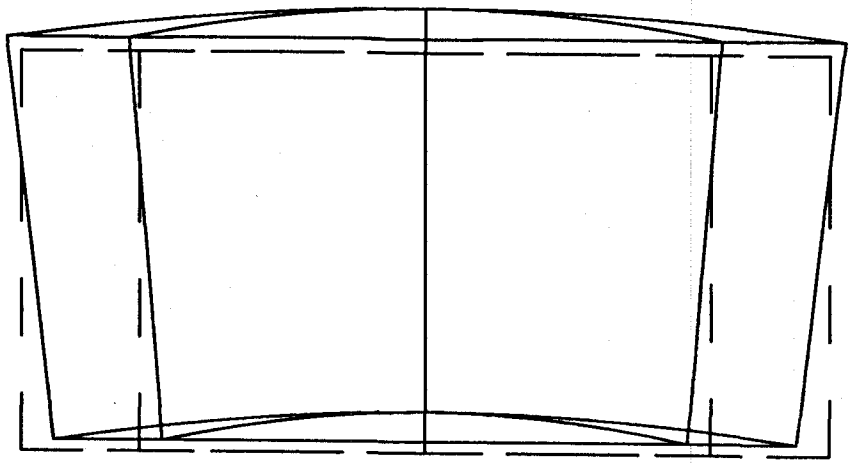
EIGENVALUE = +3.136E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 28.187$ KHz

ABAQUS

U
MAG. FACTOR = +4.2E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1 3

Mode analysis of test mass

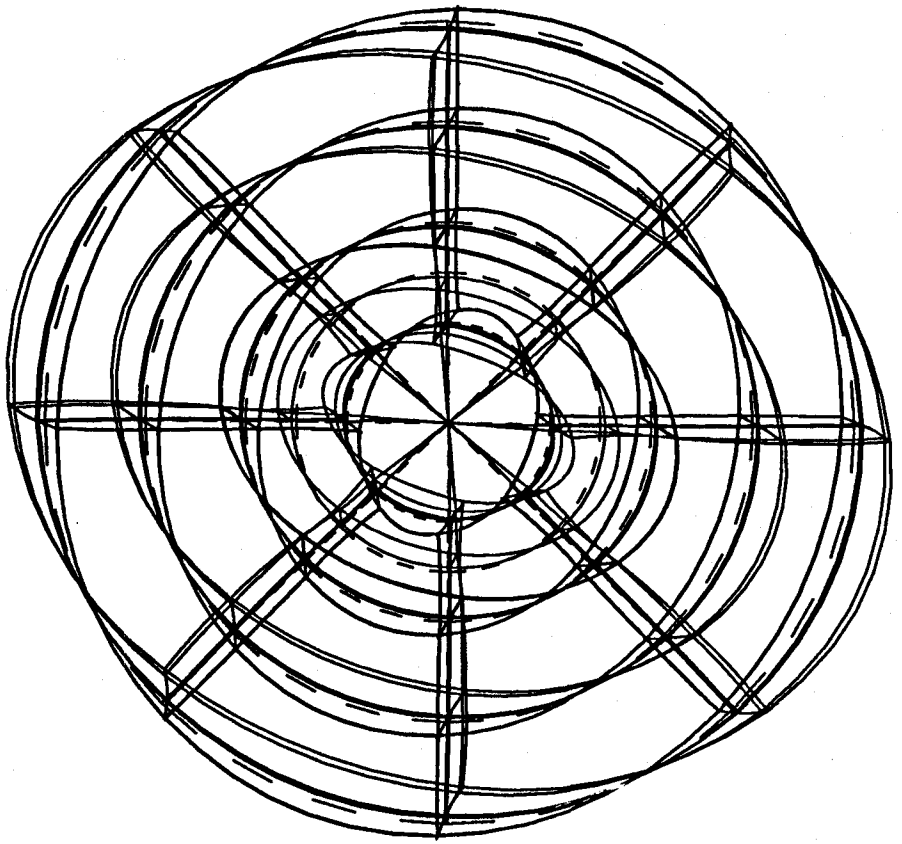
EIGENVALUE = +3.136E+10

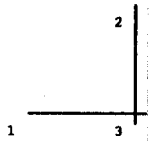
ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 29.386$ kHz

ABAQUS

U
MAG. FACTOR = +7.1E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

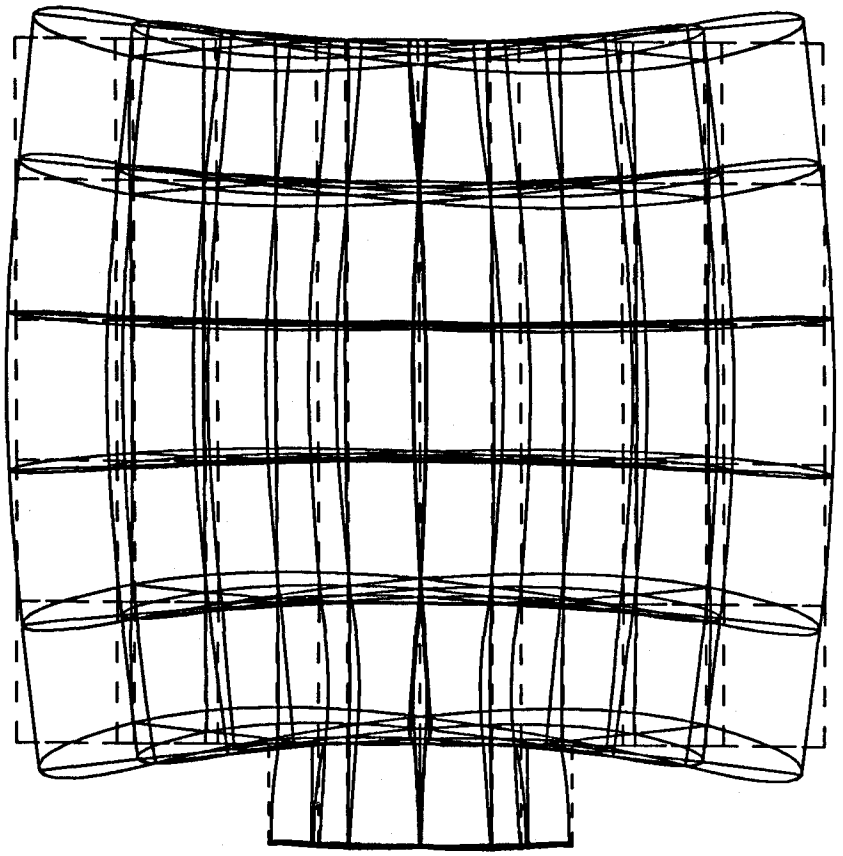
EIGENVALUE = +3.409E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 29.386$ KHz

ABAQUS

U
MAG. FACTOR = +4.4E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



Mode analysis of test mass

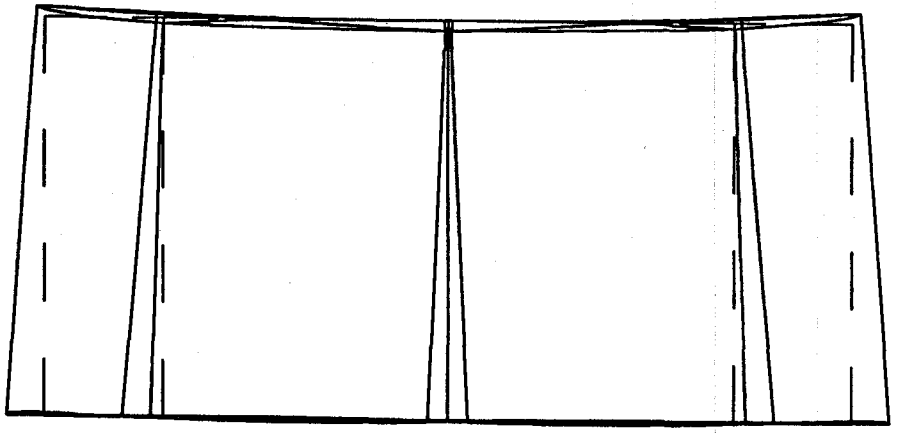
EIGENVALUE = +3.409E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 29.386$ KHz

ABAQUS

U
MAG. FACTOR = +5.4E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1 3

Mode analysis of test mass

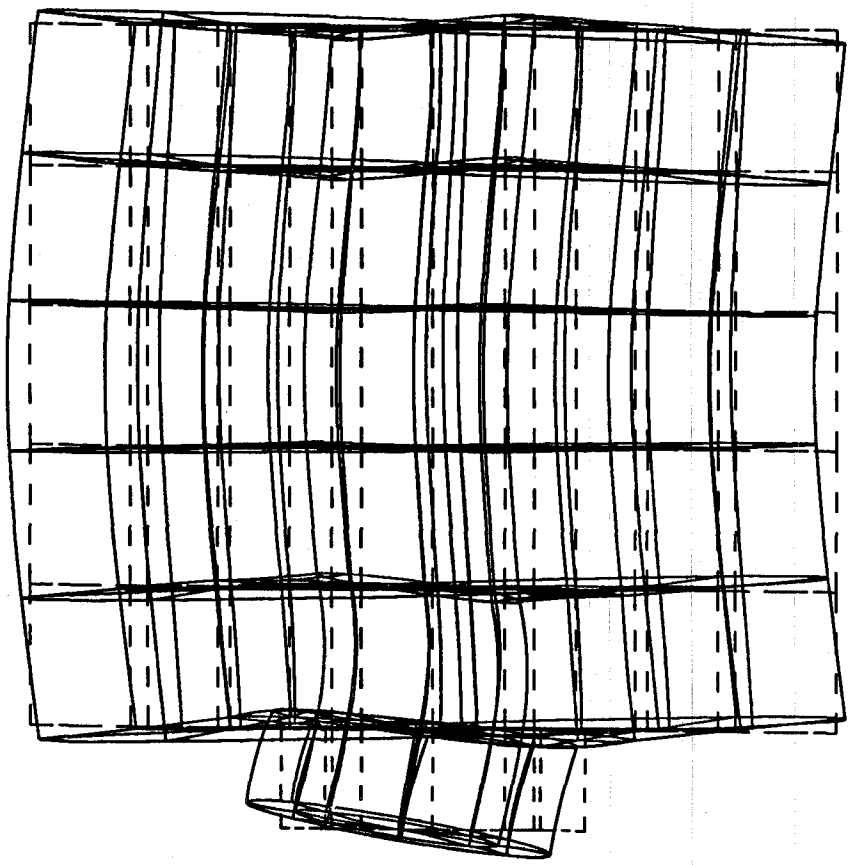
EIGENVALUE = +3.409E+10

ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

$\lambda = 31.681$ KHz

ABAQUS

U
MAG. FACTOR = +5.6E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



Mode analysis of test mass

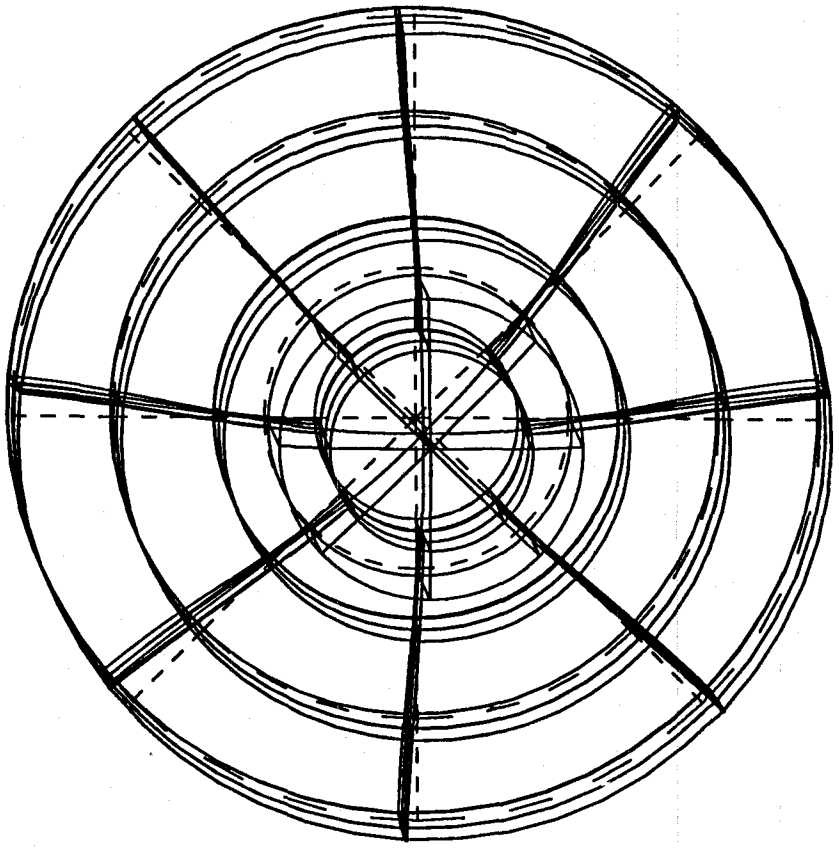
EIGENVALUE = +3.962E+10

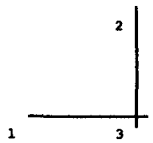
ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 31,681$ kHz

ABAQUS

U
MAG. FACTOR = +5.1E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH





Mode analysis of test mass

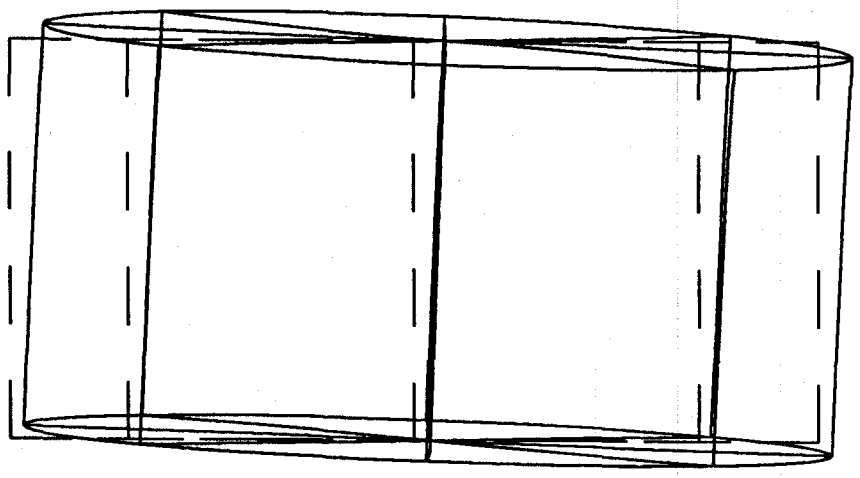
EIGENVALUE = +3.962E+10

ABAQUS VERSION 4-9-1 DATE: 14-NOV-91 TIME: 17:16:50

$\lambda = 31.681$ kHz

ABAQUS

U
MAG. FACTOR = +2.2E-01
SOLID LINES - DISPLACED MESH
DASHED LINES - ORIGINAL MESH



2 1
3

Mode analysis of test mass

EIGENVALUE = +3.962E+10

ABAQUS VERSION 4-9-1 DATE: 15-NOV-91 TIME: 15:30:59

Table of Contents

Introduction 1
Description 1
A simple plot 2
Facilities within the Command Interpreter 4
Variables 7
History 9
Changing Key-Bindings 12
Talking to the Operating System 14
Macros 15
DO, FOREACH, and IF 19
Help 20
Save and Restore 20
Vectors and Arithmetic 21
Overloading 24
Useful Macros 26
More Examples of Macros 32
Commands and Keywords 37
Glossary 38
- ANGLE 40
- APROPOS 40
- ASPECT 41
- Arithmetic 41
- AXIS 43
- BOX 43
- CHDIR 44
- CONNECT 44
- CONTOUR 44
- CTYPE 45
- CURSOR 46
- DATA 47
- DEFINE 47
- DELETE 49
- DEVICE 49
- DO 53
- DOT 53
- DRAW 53
- EDIT 53
- Environment Variables 54
- ERASE 55
- ERRORBAR 55
- EXPAND 55
- FFT 56
- FOREACH 56
- FORMAT 56
- GRID 56
- HELP 57

HISTOGRAM 57
HISTORY 57
IDENTIFICATION 57
IF 58
IMAGE 58
KEY 59
LABEL 60
LEVELS 61
LIMITS 61
LINES 62
LIST 62
LOCATION 63
Logical Operators 63
LTYPE 64
LWEIGHT 64
MACRO 64
MINMAX 65
NOTATION 66
OVERLOAD 66
POINTS 66
PRINT 67
PROMPT 67
PTYPE 68
PUTLABEL 69
QUIT 69
RANGE 69
READ 70
RELOCATE 71
RESTORE 71
RETURN 71
SAVE 72
SET 72
SHADE 74
SHOW 75
SORT 75
SPLINE 75
TERMTYPE 75
TICKSIZE 76
USER 76
Variables 77
VERBOSE 77
VERSION 78
Whatis 78
WINDOW 79
WRITE 79
XLABEL 80
YLABEL 80
Appendix I: How The Command Interpreter Works 81

Token Generation 81
Peculiarities of the Grammar 82
The Macro Processor 83
The DO, FOREACH, and IF commands 84
Examples 84
Appendix II. The Stdgraph Graphics Kernel 86
Graphcap. 86
The Binary Encoder. 90
Examples 91
Cursors 94
Colours 95
Writing a New Graphcap Entry 95
Raster Devices 96
Compiling Graphcap 98
Appendix III. Calling SM from Programmes 99
Appendix IV. The SM Grammar 103
Appendix V. Two-Dimensional Graphics 110
Filecap. 110
Appendix VI. Termcap 114
Appendix VII. Adding New Devices, and Porting to New Machines 116
Adding New Devices 116
Porting to New Machines 120
Appendix VIII. Macro Libraries 121
Appendix IX: Tips for Mongo Users 125
Differences from Mongo 125
The READ OLD command 126
The compatibility macro 126
Appendix X. The Available Fonts 128
Index 134

SM

Robert Lupton and Patricia Monger† 25th September 1991

Introduction

This document starts with a discussion of the command interpreter's main features. The second part is a description of each of the commands which are currently available, followed by appendices giving technical details and an appendix giving the font tables.

SM is still evolving slowly, and this documentation may not be true, helpful, or complete. In order of increasing plausibility, information may be obtained from this document, the HELP command, the authors, and the source code. RHL is prepared to guarantee that the executable code has not been patched.

If you find bugs, (reasonable) features that you want, wrong documentation, or anything else that inspires you please let us know. At least under Unix the macro `gripe` should be a convenient way to send us mail. Please also send us any clever macros that you would like to share.

Description

SM is an interactive plotting programme, with a powerful and flexible command language. The plot data may be defined to SM in a number of ways. There is also a powerful mechanism for defining and editing plot **macros** (sets of SM plot commands that are defined and invoked as plot "subprogrammes").

The features of SM are described fully in the next few sections, but let us start with a description of how to produce your first SM plot. Before you start, notice that SM is case sensitive. Keywords may be typed in lower or uppercase (as we do in this manual), but we would recommend using lowercase. It is in fact possible to change the meanings of lowercase keywords, but this can be confusing. If you are interested, see the section on "overloading". See 'uppercase' in the index if you really want to use your shift key.

† lupton@uhifa.ifa.hawaii.edu and monger@radio.astro.toronto.edu

A simple plot

Let us assume that you have a file called `mydata`, which looks like this:

```
This is an example file

1  1   1
2  4   8
3  9   27
4  16  64
5  25  125
6  36  216
7  49  343
8  64  512
```

SM has a history mechanism, so first type `DELETE 0 10000` to tell SM to forget any commands that it has remembered. Then choose a device to plot on. You do this with a command like `dev tek4010`. If you don't know what to call your terminal, use the `LIST DEVICE` command, ask some local expert, look at the description of `DEVICE`, or (if desperate) read Appendix II. You'll know that you have succeeded if typing `BOX` draws a box.

You should now have successfully chosen a graphics terminal. To actually plot something, use the following set of commands. The text after the # is a comment, you don't have to type this (or the #).

```
DATA mydata          # Specify desired datafile
LINES 3 100          # Choose which lines to use
READ i 1             # Read column 1 into 'i'
READ ii 2            # Read column 2 into 'ii'
READ iii 3           # Read column 3 into 'iii'
LIMITS i ii          # Choose limits, based on i and ii
BOX                  # Draw the axes
PTYPE 4 0            # Choose square point markers
POINTS i ii          # Plot i against ii
CONNECT i ii         # and connect the points
XLABEL This is i     # Label the X-axis
YLABEL This is ii    # And the Y
```

You should now have a graph. If you had wanted to plot the third column instead of the second you could have typed `LIMITS i iii POINTS i iii` instead. And of course you could plot `ii` against `iii` as a third alternative. You were not limited to only use squares as markers or solid lines to connect them - see `PTYPE` and `LTYPE` for details.

If you want a logarithmic plot, SM makes that easy for you. You can take logs of a vector using the `LG` (or `LN`) commands on vectors; try it - `SET x=1,10 SET y=x**3 set ly=LG(y) LIMITS x ly CON x ly box`. You might have wanted the axes to reflect the fact that you had logged the y axis. The `TICKSIZE` command allows you to do this, and this is in fact the commonest use of it. Try `TICKSIZE 0 0 -1 0`, and then repeating the x-y plot.

What if you want hard copy of your hard-earned graph? There is a command (actually a macro) called `playback` which will repeat all the commands that you have typed. Type `ERASE` to clear the screen, then `HISTORY` to see the commands that you have issued. You probably don't want the `ERASE` command to be repeated, so type `DELETE` to delete it*. If there are any other mistakes use `DELETE m n` to delete the lines m to n containing them. Now type `playback` and your plot should reappear. But we wanted a hardcopy, so type `dev laser lqueue` (or whatever your friendly Guru recommends as a hardcopy device), then `playback`. This time, those plotting commands will appear on the laser printer not your terminal. To make them actually appear, type `hardcopy` or issue another `dev` command. Be sure to say `dev tek4010` (or whatever device you chose) before you read any more of this document. It is possible to edit the playback buffer, rather than simply deleting lines from within it. The section on 'examples' describes how to do this.

In fact, the same plot could have been produced from a data file which just contained the first column. After saying `READ i 1`, you could have said `SET ii = i*i SET iii = i**3` and proceeded from there, or even skipped the file altogether by saying `set i = 1,8` instead of `READ`ing it at all. Such possibilities, and a good deal more, are described in greater detail in the rest of this manual.

What we just did was to define a simple 'macro', in this case the special one called `all` which `playback` manipulates. A more explicit use of a macro would be to define a macro to square a vector, that is to square each element of a vector. To do this, say

```
MACRO square 2 { SET $1 = $2*$2 }
```

So to calculate the vector `ii` we could now say

```
square ii i
```

which is the same as saying

```
SET ii = i*i
```

So now that you have met macros, how do you save them? The simplest and least reliable way is to use SM's history, and hope that the next time that you use SM it remembers the `MACRO` command that you used to define `square`, so you can re-issue it. (Try exiting SM, then starting it up again and typing `HISTORY`, then `^nnn` where `nnn` is the number which is next to the desired command in the resulting list.)

A brute force way is to say `SAVE filename` which will save almost all of your SM environment, to be recovered using the `RESTORE filename` command at some later time, or later SM session. Specifically, `SAVE` will save all your macros, variables, and vectors, along with your history buffer. This is a very convenient way in practice but it does mean that you tend to carry around lots of long-forgotten macros, variables, and vectors.

---

* There is a macro `era` defined as `DELETE HISTORY ERASE` that wouldn't have appeared on the list in the first place, similarly `lis` is like `HISTORY` but won't appear on the list of commands. As an alternative, you can use the macro `set_overload` to make lowercase `erase` the same as `DELETE HISTORY erase`, along with a number of other changes. This could be confusing for neophytes! See "overloading"

Another way is to write the macros to a disk file, using the MACRO WRITE command (see 'Macros'). Then you can retrieve your macros with MACRO READ. You should note that your macro all will simply be a macro - to put it onto the history list say DELETE 0 10000 WRITE HISTORY all. (Of course, you could write a macro to do this for you). Maybe saving your playback buffer is something better done with SAVE, which will restore your playback buffer, while preparing files of useful macros is a use for MACRO READ. Once the idea of macros gets into your blood, you can of course use an editor to create your own files of macros, to be read with MACRO READ.

Facilities within the Command Interpreter

This is a guide to the use of SM variables, the macro processor, the help command, and the history facilities. The vector arithmetic and plotting facilities are described below. Various examples are scattered throughout the text, to give some guidance on the use of SM's capabilities.

Perhaps the most important thing to know is how to escape from SM. If you have a prompt, simply type QUIT †. If you are running some command, try ^C to get a prompt. Most commands will eventually return control to the keyboard following a ^C. In addition, the parser is reset, and the input buffer cleared. Sometimes ^C leaves a } on the buffer if it thinks that it'll help get back to the prompt, which can generate an irrelevant syntax error. Occasionally it can still be confused – try typing a few characters and maybe a } . When you have interrupted SM with a ^C, a macro called error_handler is executed, if it is defined. The one that we provide does things like setting the expansion back to 1, and resetting any window commands that you might have issued, and then prints a message handler... to tell you that it's done its work. If you don't like this, see 'private initialisation' in the index for how to get your own handler loaded automatically. If you make a mistake, and SM notices a syntax error, it'll print a message indicating where you were and which macro you were running. It is possible for the wrong macro to be reported (if SM has finished reading the macro before detecting the error), in which case you'll be told that the error occurred in a macro that called the offender. Setting VERBOSE (*q.v.*) to 3 or 4 provides a more direct way of finding the true location of the error. In addition, the usual interrupt capabilities of your operating system will work under SM, with a couple of quirks. Under Unix, in case of emergency, type ^\, and SM will ask you if you want to return to the prompt, and if you don't it'll offer a core dump, and then exit. As usual, typing ^Z (from the C-shell) will interrupt the process, which may be restarted later. Under VMS, ^Z will interrupt SM, and return you to the command interpreter (DCL). Typing CONTINUE will then allow you to restart SM *.

---

† or q which is a macro defined as DELETE HISTORY QUIT. This will exit SM just the same, but the quit won't appear on your history list, waiting to be playedback accidently. Actually, q will query you before quitting

* If you are using VMS, you may prefer to use ^Y as your interrupt character. A suitable set of key definitions is in a file called **maps_vms.dat** in the SM macro directory. It may be read with the READ EDIT command, and this may be done automatically in your startup macro, usually by examining the variable edit in your **.sm** file.

If SM is running in a SPAWNed sub-process, then ^Z will reATTACH you to its parent. To continue SM, use the DCL ATTACH command. We strongly suggest that you learn how to do this, it makes life much easier – all you have to do is SPAWN a process from DCL and start SM from there. Do check with your VMS system manager to ensure that you have the right quotas for SPAWNing (Process limit must be at least 3, because SM will use one for itself and one for hardcopies). An especially simple way to do all this is is to use the command file **kept_SM.com** in the main SM directory. It'll handle the spawning and attaching for you.

Another fact to bear in mind is that the characters ^, **$**, and # are special, as ^ is used by the history system, **$** introduces a variable, and # starts a comment. The special meanings of all of these characters except ^ can be turned off by preceding them with a \. To type a ^, use the quote_next character (initially ^Q or ESC-q) to quote the ^; *i.e.* type `ESCq^`. † A **\n** is interpreted as a carriage return, and a \ as the last character on a line escapes the newline, so that the line and the one following it are treated as one long line. A \ preceding any other character (except a "; see next paragraph) is simply a \. This character is used to set font types in the `LABEL` commands, so it has no special meaning to the command interpreter, which simplifies the entering of strings for `LABEL` commands.

A further problem is that symbols such as +, -, *, and / are used to separate words, which is what you want for mathematics, but *maybe not* what you had in mind for filenames. Enclosing a word in double quotes turns off all special meanings except ^ ; an embedded " may be escaped with a \. On the subject of meanings of characters, note that SM is case-sensitive. It will accept keywords in either upper or lower case, but this is a special dispensation on its part. If you insist on typing in uppercase say `load uppercase` when you first start SM, or put the line 'uppercase 1' in your **.sm** file. Furthermore keywords may not be abbreviated. This is not a great hardship as it is easy to define macros which make the minimum abbreviation a synonym for the full command. Many such macros are predefined for you when you first use SM; see the section on 'Useful macros' for details. In particular, certain common abbreviations of commands have been predefined by the SM startup file.

Every time that SM is started, it looks for an environment file called **.sm** in your current directory, or, failing that, in your home directory. This file consists of names of variables and their values. From # to the end of a line is taken to be a comment. An example file would be (the filenames are written in Unix)

```
# I'm a comment line
    fonts       /users/sm/fonts.bin
    graphcap    /users/sm/graphcap
    help        /users/sm/help/
    macro       /users/sm/macros/
    name        Robert    # Or alternatively 'Dr.  Lupton'
```

The `fonts` file contains the SM fonts (in a binary form), the `graphcap` entry is used to define the file used to describe graphics terminals (See Appendix II on 'Stdgraph'),

† In fact you can rebind any character to replace ^, this is discussed under history.

`help` is the directory used by the help command, `macro` is the default directory where macros reside, and `name` is what SM will call you (you can put spaces into your name by using underscores, *e.g.* `My_Lord` will be referred to as `My Lord`). You can access entries in the environment file yourself, as described in the section on variables. See also the section on 'useful macros' to see how entries in the **.sm** file are used to influence the behaviour of SM, or consult your local expert. It would probably be a good idea to borrow someones **.sm** file when you first use SM. For more detail, and further special entries, see the section on **.sm** files. The name of the **.sm** file can be specified on the command line as "`-f name`". VMS users should ensure that SM has been installed as a foreign command to take advantage of this capability.

SM then tries to read in any macros in the file **default**, in the directory **macro** or failing that, the current working directory, and attempts to execute the macro `startup` if it exists. If `-m filename` appears on the command line, this is taken to be the name of another file of macros and these are read, and the eponymous macro is executed (after any pre- or suf- fix has been removed. For instance if you start SM with the command `smongo -m /home/tst.m`, it will first read the file `/home/tst.m`, and then attempt to execute the macro `tst`). * Anything left on the command line is treated as if it had been typed at the prompt, for example `sm restore vital.save` will start by `RESTORE`ing from the file `vital.save` (see `RESTORE` if you want to know what this means). The `-m` option is not really a good way to personalise SM. The `startup` macro discussed under 'useful macros', which is run everytime that you start SM, looks for a directory `macro2` in your **.smongo** file, and if it is there reads a file **default** from it, and executes the macro `startup2` which it expects to find there. On case-insensitive operating systems, such as VMS, you may need to quote the command line to prevent it being translated to upper case.

SM then attempts to read a set of history commands from a file in the current working directory, passes control to the input routine and issues a prompt. The file is given by the entry `hist_file` in your **.sm** file, and if it isn't present then no history will be remembered. You are then able to type commands, as many as will fit on one line †, and use the features described below.

You can use a combination of these features to run SM in 'batch' mode. If you had a history file that you just wanted to run, then you could start SM, say `playback`, and quit. You could have a macro called `batch` in `batch.m` that did just that, and say `sm -m batch.m` to execute it. In fact, you don't even need your own macro as one is pre-defined for you so `sm batch` is sufficient. You could write your own macros along this lines to do more complex tasks, but you should be careful *not* to include any commands that don't appear on the history list (such as `playback` itself), as they'll delete commands from the history list.

---

* Under VMS, SM must have been installed as a foreign command for this to work, and it must *not* have been linked with the debugger

† Occasionally a <CR> is required by SM, so putting two commands on one line will give a syntax error. The reason is given in Appendix I, the fix is either to use two lines, or else to put an explicit carriage return at the appropriate point with a \n.

For completeness, we should mention the other four command line flags, `-l logfile`, `-s`, `-S`, and `-v#`. If you specify a logfile with `-l` everything that you type at the keyboard is copied into the logfile (except editing commands). The `-s` (for 'stupid', or 'silent' or 'suppress') flag disables the history and command line editor, `-S` is like `-s` but it also suppresses the prompt. This is useful if SM is being run from inside another programme, via a pipe (VMS: mailbox), or on a *very* stupid terminal. If you want to set a particular value of verbose, use `-v`, for example `-v-3` is equivalent to the VERBOSE -3 command given interactively.

Variables

SM maintains a set of variables which are defined with the statement

```
DEFINE name value
```

or

```
DEFINE name { value_list }
```

or

```
DEFINE name ( expression )
```

where `name` must consist of digits, letters and '_' (but must not start with a digit), and may be a keyword. Variables are *string*-variables, and are not primarily designed for doing arithmetic (that's what vectors are for).

`Value` may be a word or a number. `Value_list` has no such restrictions and may contain many words. Note that due to the presence of the { } , variables are not expanded in `value_list`, whereas they are in `value`. In fact, the list can be delimited by <> rather than { } ; see `DEFINE` for details. The expression in the `DEFINE` that needs one should be a scalar; if it is not, the first element of the vector will be used and you will be warned, if `VERBOSE` (*q.v.*) is one or greater. Expressions are further discussed under 'Vectors and Arithmetic'. The special variable "date" expands to give the current time and date, – try typing `echo $date`. Each time SM reads `$name` it replaces it by its value, considered as a character string. For example,

```
DEFINE hi hello
WRITE STANDARD $hi
```

will print `hello`. This expansion is done before even the lowest level of lex analysis, so if a command is attempting to read a value it is possible to give it the name of a SM variable. An example would be the `XLABEL` command, which writes a string as the x-axis label of a graph,

```
DEFINE name Aelfred
XLABEL My name is $name
```

will invoke the `XLABEL` command, and write `My name is Aelfred` below the x-axis. (Incidentally, `DEFINE Aelfred Aethelstan YLABEL $$name` will write `Aethelstan` as the y-axis label, which can be handy in macros. The use of the double $$ indicates to SM to do a double translation, as it first expands to `$Aelfred` which then expands to `Aethelstan`).

A variable can be deleted by `DEFINE name DELETE` so for example the macro

```
MACRO undef 1 { DEFINE $1 DELETE }
```

invoked as

```
undef name
```

will undefine the variable `name` (see the section on macros if you are confused).

There are also three special `values`, :, |, and ?, where : means 'get the value of this variable from the environment file', | gets the value of an internal variable (such as the current expansion), and ? means 'read the value from the keyboard'. You can specify a prompt to be used, again see `DEFINE` for details. A list of the | variables is given in the section on `DEFINE`. So using the example **.sm** environment file listed in the previous section of the manual, `DEFINE name :` will define `name` to be `Robert`, `DEFINE angle |` will give the last value set by the `ANGLE` command, and `DEFINE datafile ?` will ask you for the value of 'datafile', which can be useful in macros. For example,

```
DEFINE noise ?  { Ring bell?  } IF('$noise' != 'n') { bell }
```

will execute the macro `bell` if you type anything but n in reply to the question 'Ring bell?'.

When writing macros, it is also sometimes useful to know if a variable has been defined. The variable `$?name` has the value 1 if `name` is defined, otherwise it is 0. For instance, there is a line

```
define term :  if($?term) { termtype $term}
```

in the startup file, to set a termtype if present in the environment file. There are also commands to read the values of variables from data files defined with the `DATA` command.

```
DEFINE name READ i
```

or

```
DEFINE name READ i j
```

will set `name` to be the i^{th} line of the file (or the j^{th} word of the i^{th} line). An example is given in the section on 'useful macros'. You can read variables from the headers of binary files (specified with the `IMAGE` command) using `DEFINE name IMAGE`, although this is only supported for a limited class of `file_type`'s (*q.v.*).

All currently defined variables may be listed with

```
LIST DEFINE [ begin end ]
```

where the optional `begin` and `end` define the range of variables (alphabetically) to be listed.

Variables are usually not expanded within double quotes or { } . If for some reason you need to force expansion within double quotes, it can be done with `$!name`. The macro 'load' discussed under useful macros gives an example of this mechanism. If you need to expand a variable, with no questions asked (and even within { }), use `$!!name`.

Sometimes you may want to terminate a variable name where SM doesn't want to, and this can be done with a trick involving double quotes. Say you are writing a macro to find all the stars redder than B-V = 1.0 in a set of data vectors, and you want to rename them with a trailing "_red", so `star` goes to `star_red`. So you

write a foreach loop,

```
FOREACH x ( U B V R I J K ) { SET $x_red = $x IF(B-V >1)}
```

Well, that won't work because SM thinks that you are referring to a previously defined variable named `x_red`, so it will complain that `x_red` is not defined. But if you write it as `$x""_red` the "" separate the `x` from the `_red` until `$x` is expanded, and then disappear, and all is well. When a variable is read, SM skips over all whitespace before the definition, and this can cause problems if you hit ^C in the middle, as the rest of the command will be thrown away. If you ever hit a ^C, and can't get a prompt, try typing any non-whitespace character.

History

It is often very useful to be able to repeat a command, or perhaps correct a mistake in what you have just typed. Ways of doing this are usually referred to as 'history', and SM has two distinct mechanisms. One is very similar to that of the Unix C-Shell, and the other allows you to edit commands using a syntax similar to the popular editor 'emacs', or a generalisation of the DCL history under VMS. If you are not familiar with Unix, emacs, or VMS don't despair; a description of the commands and how to invoke them follows in this document. Both of these mechanisms are implemented by the routine which reads input lines. As each line is sent to the parser, it is copied onto a history list. This list may be printed with `HISTORY`, and the commands may be re-used by referring to them by number, as `^nn`, or by a unique abbreviation, as `^abbrev`. In addition, the last command may be repeated by using `^^` and the last word of the last command by `^$`. These symbols are expanded as soon as they are recognised (see examples, or experiment), and are then available for modification by the editor. Sometimes a `^string` will retrieve a command beginning `string`, but not the one that you want. In this case, type `^TAB` (two characters, ^ and `TAB`) to continue the search for the next-most-recent command beginning `string`, and so on until you find the one that you want. This will only work until you type a carriage return, when everything starts again from scratch. Some people really don't want ^ to be their history character, either because they're used to something else (such as !), or because they want to type lots of real ^s (*e.g.* you are using TEX-style strings); if this describes you, rebind them – see the next section. If you are considering the history list as a sort of programme to be repeated you may think that `HISTORY` lists the commands in the wrong order; if so use `HISTORY -`.

For example, if I type:

```
PROMPT @
echo I am SM
HISTORY
```

SM will set the prompt to be @, replace the macro `echo` by its value `WRITE STANDARD` and print

```
I am SM
```

and then

```
3 HISTORY
2 echo I am SM
1 PROMPT @
```

(The actual numbers will be different, depending on what other commands you have executed, and also because SM may have read a history file. In that case there'll be many more commands on the list, but no matter.) If I then type

^2 <CR> (that is ^2 not control-2)

the screen will look like

```
@ echo I am SM
I am SM
```

as if I had just typed it in (@ is the prompt) . Typing

^^ (Yes, ^$) <CR>

will now result in SM printing (truthfully)

```
I am SM (Yes, SM )
```

It is possible to delete commands from the history buffer with the **DELETE** command. If the command is given with one or two arguments, then the specified range is deleted (but their numbers are not re-used). If no arguments are given, the last command on the buffer is deleted, and its number is released to be re-used. In other words, the command **DELETE** will delete first itself, and then the previous command from the history list. The command **DELETE HISTORY** only removes itself from the history list, and several of the common commands are defined as macros which use it, for instance **dev** is defined as **DELETE HISTORY DEVICE**. This means that the command will not appear on the history list, to confuse you when you do a playback. But if you now innocently use **dev** in a macro, that macro won't appear on the list either. Still worse, if you use **dev** twice in one macro, the previous command will be deleted as well which could be quite confusing.

By default only 80 lines are remembered, and as you continue typing earlier ones fall off the list.Because the history buffer is also used to compose complex commands, this limit can be aggravating. You may be able to defeat this by putting many commands on each line (you may have to use **\n** to terminate label commands explicitly) or by writing macros. Alternatively you can define a longer history buffer when you start SM by including an entry **history** in your environment file which gives the number of commands to be remembered. Incidently, it is the total number of commands that matters, not the total range of history numbers present.

The editor allows you to modify commands, either as you type them or as you retrieve them from the history list. The various editing commands may be bound to keys of your choosing, but the default bindings are given in this list of possible commands:

^A	Go to start of line.
^B	Go back one character. (Equivalent to ←).
^C	Interrupt (as usual).
^D	Delete character under cursor.
^E	Go to end of line.
^F	Go forward one character. (Equivalent to →).
^H	Identical to ^? (DEL). Delete character to left of cursor.
^I (TAB)	Insert spaces up to the next tab stop.
^J (LF)	Equivalent to ^M.
^K	Delete to end of line. The deleted string is stored, and may be restored using ^Y, repeatedly if so desired.

^L	Redraw the current line.
^M (CR)	Send line to be executed. Some terminals seem to replace ^M with a linefeed (^J), thereby making it impossible to make SM obey you. We therefore make ^@ equivalent to ^M for emergency use. (This is control-space on many terminals).
^N	Get the next command on the history list, if it exists (see ^P). (Equivalent to ↓).
^O	Execute the previous command on the history list. Equivalent to ^P^E^M.
^P	Get the previous command on the history list, if it exists (see ^N).(Equivalent to ↑).
^Q	Quote next character; Turn off any special significance to the editor. ^Q is often used by the terminal, so we have defined ^[-q (that is escape followed by q) as an alternative.
^T	Toggle insert/overwrite. By default, characters are inserted before the cursor. If overwrite is set, they replace the character under the cursor. Note that ESC-u will not correctly restore words deleted with ESC-d in overwrite mode.
^U	Delete from the cursor to the start of the line.
^V	Go forward 5 lines when editing macros.
^W	Delete the previous word. Identical to ESC-h
^Y	Insert the string most recently deleted with ^K after the cursor.
^Z	Return to the operating system, without killing SM. (Under VMS, if you are running SM in a spawned subprocess ^Z will attach you to DCL. Otherwise, SM returns you to DCL.)
^? (DEL)	Equivalent to ^H.
ESC-<	Go to the first line of a macro, or the oldest history command.
ESC->	Go to the last line of a macro, or the most recent history command.
ESC-g	Go to a given line of a macro, or a given history command. You'll be prompted for the line number, if you change your mind you can get out with DEL or ^H.
ESC-q	Quote the next character, turning off any special significance to the editor. Identical to ^Q.
ESC-v	The opposite of ^V, go back 5 lines when editing macros.
ESC-y	Like ^Y, except that it gets older deletions, cycling back through a collection of (currently 5) deleted lines.

Some ESC-letter combinations are available which operate upon complete words. A word is defined as a whitespace delimited string, so 2.998e8 is a perfectly good word. In addition, it is possible to undelete words that have been deleted with an ESC-d or ESC-h.

ESC-b	Go to the start of the previous word.
ESC-d	Delete to the beginning of the next word.
ESC-f	Go to the beginning of the next word.

ESC-h	Delete back to the beginning of the current word. The same as ^W.
ESC-u	Restore the last word deleted, putting it before the cursor. Further ESC-u's will restore more words. When no more are available, the bell is rung.

Any printing character is inserted before the cursor (unless overwrite has been set with ^T). Illegal characters ring the terminal bell. If you insert a non-printing character on a line, the cursor may get confused. It may also get confused if you edit a macro that was created with a text editor, and which contains real tab characters.

If ever you are stuck at the command interpreter, and you want to send a signal to the operating system (e.g. a ^Y to DCL), but SM is catching the key and using it for its own purposes, the easiest thing to do is to define a macro such as `MACRO aa { aa }` , and then run it. While it is running (*i.e.* until you type ^C) keys should have their usual functions.

Changing Key-Bindings

As mentioned above, it is possible to redefine the meanings of keys to the history (and macro) editor. The command `EDIT keyword key-sequence` will make typing that sequence of keys correspond to the command `keyword`. For example, to make ^R redraw the current line, you could say `EDIT refresh ^R`. The keyword can be any in the list below, or any single character. Each character in the key-sequence can be a single character, `^c`, or `\nnn` where `nnn` is an octal number. Alternatively, `READ EDIT filename` will read a file specifying the new bindings which has two lines of header, followed by pairs of `keyword key-sequence`. Lines starting with a # are comments. An example is the file for VMS users given below.

On a somewhat similar topic, the `KEY` (*q.v.*) command may be used to define a key to generate a string. See the end of the section on macros for how this works.

All the current key definitions may be listed using `LIST EDIT`, including the `KEY` definitions. The names of operators, and their default bindings, are given in the following table:

start_of_line	^A	previous_char	^B
delete_char	^D	end_of_line	^E
next_char	^F	illegal	^G
delete_previous_char	^H, DEL	tab	^I
kill_to_end	^K	refresh	^L
carriage_return	^J, ^M, ^@	next_line	^N
insert_line_above	^O	previous_line	^P
quote_next	^Q, ESC-q	toggle_overwrite	^T
delete_to_start	^U	scroll_forward	^V
delete_previous_word	^W, ESC-h	exit_editor	^X
yank_buffer	^Y	attach_to_shell	^Z
first_line	ESC-<	last_line	ESC->
escape	\034	previous_word	ESC-b
delete_next_word	ESC-d	next_word	ESC-f
goto_line	ESC-g	undelete_word	ESC-u
scroll_back	ESC-v	yank_previous_buffer	ESC-y
history_char	^		

A simple example of a bindings file for a hardened VMS user might be

```
# This is a set of DCL-ish key maps for SM
#              name  key
toggle_overwrite     ^A
start_of_line        ^H
delete_previous_word ^J
yank_buffer          ^R
attach_to_shell      ^Y
```

Note that that's ''^A'' not control-A. It could just as well have been written \001. We need a new character for **yank_buffer** now that ^Y is otherwise engaged, and I have chosen ^R. You should be warned that some terminal protocols map ^M to ^J, so this use of ^J could render you unable to issue commands. As mentioned above, in an emergency ^@ can be used instead of ^M.

When SM is started, or whenever the **termtype** command is used to change terminals, the arrow keys are bound to the commands **previous_line**, **next_line**, **previous_char**, and **next_char**. For terminals such as a Televideo-912, which uses characters such as ^K for arrow motion, these can supersede the previous meanings (in this case **kill_to_end**); The only fix is to use the **EDIT** or **READ EDIT** command to get what you want, probably within a macro.

If you want to use ' as your history character instead of ^ you need to say **edit history_char ' edit ^ ^**. If you try to use a character special to SM such as ! this won't work (you'll get a syntax error), but **edit history_char "! "** will. The space

after the ! is required. Of course, you can put these into a file which `read edit` reads in your `startup2` macro:

```
# Change the history character
#      name   key
history_char  !
^             ^
```

Because this particular change is so common, it's possible to specify that ' be your history character simply by including a line `history_char '` in your **.sm** file (or you can choose your own character. Choosing 0 has the effect of using the default, ^).

SM needs to know something about the terminal that you are using, so as to run the history/macro editor. This is entirely separate from the problem of describing the terminal's graphics. It will try to discover what sort of terminal you're on by using the value of `term` from your **.sm** file,or failing that the value of the environment variable `TERM` (Unix) or the logical variable `TERM` (VMS). A `term` entry of `selanar -21` is equivalent to a `TERMTYPE selanar -21` command. You can also use the `TERMTYPE` command directly. SM then uses the terminal type specified to look up its properties in the termcap database – which is described in Appendix VI for the curious. You can also use `TERMTYPE` to specify the size of the screen, or to turn off SM's idea of where the cursor is. On some terminals, you can only send a cursor to an absolute position and this is chosen to be the bottom of the screen. This is not what you want for, e.g., a VT240 as it will lead to your graph scrolling off the screen. The use of a negative screen size to `TERMTYPE` will disable this cursor motion, but will also make editing lines slower. If a line of your graph is being deleted when the SM prompt appears, you may need to use `TERMTYPE dumb`.

Talking to the Operating System

Any line from ! to the newline is passed to your shell (DCL, csh, or whatever you use).* For example, `!ls` or `!directory` will list the current directory. It is also possible to change the directory that SM uses to look for data or macro files with the `CHDIR` command - for instance `CHDIR "../more_data"`.† If a directory name starts with '~', `CHDIR` replaces the '~' with your home directory. This is the only place that '~' is treated specially, for instance it is not interpreted by the `DATA` command. Because directory names often contain mathematical characters such as [or /, it is wise to quote the directory, or use the macro `cd` which quotes it for you.

---

* The first of these commands in a SM session may be rather slow under VMS, as we have to spawn a subprocess.

† Unfortunately, this is currently not available to VMS users

Macros

In SM, it is possible to define sets of plot commands as "subprogrammes", which can be used just like a plot command, to generate a standard plot. These plot **macros** allow variables (e.g. name of the data file, plot label or limits, etc) to be supplied at execution time.

You can also bind commands to keys to save typing; for example I usually bind 'cursor' to the PF1 key of my terminal. Such keyboard macros are discussed under **KEY** and at the bottom of this section.

The macro facility consists of commands to define macros, delete them, write them to disk files, read them from disk files, delete all those macros defined in a specified disk file and list all currently defined macros. In addition, the help command applied to a macro prints out its definition. It is possible to pass up to 9 arguments to a macro, referred to as $1, ... , $9, and in addition $0 gives the name of the macro. While macro arguments are being read they are treated as if they are in sets of { } , except that variables *are* expanded. If you want to include a space in an argument, enclose it in quotes. If the number of declared arguments is 10 or more, the macro is treated as having a variable number of arguments, see the discussion of how macros are used.

A macro is defined by the statement

```
MACRO name nargs { body-of-macro }
```

where **body-of-macro** is the statements within the macro, and may be up to 2000 characters long. Macros defined using an editor on a file may be arbitrarily long. If **nargs**, the number of arguments, is 0 it may be omitted. Macros may also be created using the **MACRO EDIT** command, which is discussed below, and which is probably easier. To define the macro in a disk file, the file format must be: the name of the macro starts in the *first* column, followed by a tab or spaces, followed by the number of arguments, if greater than 0, followed by commands, followed by comments if any. The next line and any necessary subsequent lines contain the macro definition *(starting in a column other than the first one)*. Any number of macros may appear in the same file, as long as the macro name is given in the first column and the definition starts in some other column. The first two blanks or tabs are deleted in continuation lines, but any further indentation will survive into the macro definition.

When a macro is invoked, by typing its name wherever a command is valid, for example at a prompt, it first reads its arguments from the terminal (if they are not in the lookahead buffer, it will prompt you for them), and defines them as the variables $1, ..., $9, before executing the commands contained within the macro. The number of arguments must be declared correctly. As an alternative it is possible to declare that a macro has a variable number of arguments by declaring 10 or more. The macro will then expect between 0 and the number declared modulo 10 arguments, all on the same line as the macro itself. (*i.e.* the argument list is terminated by a newline, which may either be a 'real' one, or an \n). The macro may find out if a particular argument is provided by using **$?** to see if the variable

is defined. For example the macro `check`, in the format in which it would appear in a file,

```
check 11 if($?1 == 1) { echo Arg 1 is $1 }\n
```

will echo its argument, if it has one. Unfortunately, we can get into trouble with `IF`'s at the end of macros, for much the same reason that `RETURN` can get into trouble (see Appendix I). The symptoms are that a macro either gets the arguments that were passed to the macro that called it, or complains that it can't handle numbered variables at all because it isn't in a macro at all. To avoid this, there is an explicit \n at the end of the macro. It is possible to redefine the values of arguments (it won't affect the values you originally specified, arguments are passed by value), or to `DEFINE` values for arguments that you didn't declare. The latter case allows you to have temporary variables, local in scope to the macro. An example is the `rel` macro, which is defined as

```
rel 2 DEFINE 1 ($1) DEFINE 2 ($2) RELOCATE $1 $2
```

which allows you to specify expressions to the relocate command. For more examples see the 'useful macros' section.

Newlines are allowed within macros, and as usual any text from a # to the next newline is taken to be a comment. If a # is needed within a macro, escape the # with a \ or enclose it in double quotes. If a macro starts with a comment the comment will not affect the macro's speed of execution. Macros starting with ## are treated specially by `SAVE` (they are not saved) and `MACRO LIST` (they are not listed if `VERBOSE` is 0).

If the macro command is given as

```
MACRO name { DELETE }
```

or

```
MACRO name DELETE
```

the macro will be deleted (you can also delete a macro from the macro editor by specifying a negative number of arguments). The name may be up to 80 characters, and must not be a keyword. If the name is already defined, it will be silently redefined. Macros may be nested freely, and even called recursively. For example, the definition

```
MACRO aa { aa }
```

is perfectly legal, but SM will go away and think about it for ever if you ever type `aa` (or at least until you type ^C.) The definition

```
MACRO zz { zz zz # comment:  not recommended }
```

is also legal, but in this case if you execute it SM will fill its call and macro stacks and complain when it grabs more space. As before, it will think about it forever. A more useful example of a recursive macro is `compatible` discussed in Appendix IX, which starts

```
IF($?1 == 0) { compatible 1 RETURN } ...
```

providing a default value for its argument. The macro `undef`, given previously, is an example of a macro with an argument.

To find how a particular macro is defined, type `HELP macroname`. For a listing of the first line of all currently defined macros, type

```
LIST MACRO
```

or

```
LIST MACRO x y
```

The optional `x` and `y` are the alphabetical (actually asciial) range of macro names to list. As mentioned above, if `VERBOSE` is 0, macros starting with `##` are not listed by this command. There is a macro `ls` defined as `DELETE HISTORY LIST MACRO` which will list macros without appearing on the history list. (Or you could overload `list`; see under overload in the index).

A related command is `APROPOS pattern` which lists all macros whose names or initial comments contain the `pattern`, for example

```
APROPOS histogram
```

would list `bar_hist` and `get_hist` as well as the abbreviations `hi` and `hist`. If you wanted to find all macros starting with a single comment character which mentioned `histogram` you could say

```
APROPOS "^#[^#] .*histogram"
```

where the double quotes prevent the #'s being interpreted as comment characters.

```
APROPOS ^[a-z]
```

will list all macros beginning with lowercase letters – this is similar to `MACRO LIST a z`, but pays no attention to the value of `VERBOSE`.

It is also possible to read macros in from disk, and in fact when SM is started, it tries to read the file **default** in the directory specified by `macro` in the environment file **.sm**. The command to read a file of macros is

```
MACRO READ filename
```

Any line with a # in the first column is treated as a comment, and is echoed to the terminal if `VERBOSE` is greater than zero. All the currently defined macros may be written to a file with the command

```
MACRO WRITE filename
```

If the file exists, it will be overwritten (under VMS, a new version of the file will be written). Macros are written out in alphabetical order. The command

```
MACRO WRITE "/dev/lp"
```

is an example of this, and writes all currently defined macros to the line printer (under Unix, that is).

The command

```
MACRO WRITE macroname filename
```

writes the macro `macroname` to the file `filename`. This command remembers which file it last wrote a macro to, and if the current filename is the same then it appends the macro to the end of the file, otherwise it overwrites it (or creates a new version under VMS) unless the `filename` is preceded by a `+`, in which case the macros will always be appended. This allows a set of related macros to be written to a file.

```
MACRO DELETE filename
```

undefines all macros which are defined in `filename`. This allows a file of macros to be read in, used and forgotten again. If this weren't possible, if you wanted to write a new definition to file **default** and wrote out all known macros, the list would contain those from the second file read in addition to **default**. The difference between this command and `MACRO macroname DELETE` should be noted. The `SAVE` command is probably a better way of saving all current macros. The format of macros on disk

is **name nargs text**, where **nargs** may be omitted if it is 0. Continuation lines start with a space or tab. See the files in the directory specified by **macro** in your **.sm** file for examples.

It is possible to define macros from the history list. The command

```
MACRO name i j
```

defines a macro **name** to be lines **i** to **j** inclusive of the history list, as seen using **HISTORY**. This macro may then be edited (see next paragraph) to enable the user to play back a set of plot commands, *e.g.* to execute the same plot on a different plot device. The opposite of this command, which places a macro upon the history list, is **WRITE HISTORY name**. Examples of these commands are the macros **playback** and **edit_hist** given in the section 'A Simple Plot'. This way of defining macros can be convenient if you have created a useful set of commands on the history buffer, and now want to save it in a macro and go on to other things. Editing the playback buffer, and then changing its name to something else (see next paragraph) is a convenient way of saving it that implicitly uses this command.

Macros may be edited, using essentially the same keys as described above for the history editor. The command **MACRO EDIT name** starts up the editor, which works on one line at a time. † The zeroth line has the format

```
0> Macro name :  Number of arguments: n
```

where **name** is the name of the macro, and **n** is the number of arguments to the macro. If this line is changed, except to change **name** or **n**, any changes made to the macro will be ignored (note that the space after **name** is required). This can be useful if you decide that you didn't want to make those changes after all. Changing **name** or **n** has the obvious effect, except that if **n** is negative the macro is deleted when you exit the editor. An empty macro is also deleted when you leave the editor (*i.e.* one with no characters in it, not even a space). The first line that you are presented with is the first line in the macro rather than this special one. Use ^N (or ↓) to get the next line, ^P (or ↑) to get the previous line. Carriage return (^M) inserts a new line before the cursor, breaking the line in the process, while ^O inserts a new line before the current line. To save the changes and return to the command interpreter use ^X. All other keys have the same meaning as for the history editor (*e.g.* ^A to go to the beginning of a line), except that ^V and ESC-v can be used to move down or up 5 lines at a time. Note that ^K and ^Y can be used to copy lines, and that the bindings can be changed with **EDIT** or **READ EDIT**.

It is sometimes convenient to define a key to be equivalent to typing some string, such as **playback** or **cursor**. This can be done with the **KEY** command, whose syntax is **KEY key string**. If you just type **KEY** <CR> you'll be prompted for the key and string. In this case, you are not using the history editor to read the key, and you can simply hit the desired key followed by a space and the desired string, terminated by a carriage return. If you put **KEY**, **key** and **string** on one line you'll probably have to quote the **key** with ^Q or ESC-q, or write the escape sequences

† you might prefer to use the macro **ed** which is equivalent to **MACRO EDIT**, but doesn't appear on the history list and, if invoked without a macro name, will edit the same macro as last time. In addition, you can list the current macro with **hm**.

out in the way used by EDIT. If this sounds confusing, here is an example. Type KEY<CR>, then hit the PF1 key on your terminal, type a space, and type "echo Hello World\N". Now just hit the PF1 key and see what happens. (The closing \N meant 'and put a newline at the end'). These keyboard macros are not generally terminal independent, but they can be convenient. Definitions for the 'PF' keys on your keyboard can be made in a terminal-independent way by specifying the key as pfn or PFn where n is 1, 2, 3, or 4. If you always use the same terminal you might want to put some KEY definitions in your private startup file (see the discussion of startup2 in the section on useful macros). The current KEY definitions are listed with the LIST EDIT command, along with the other key bindings.

DO, FOREACH, and IF

Related to the macro facility are the DO and FOREACH commands. IF is included here as a flow-of-control keyword. The syntax for a DO loop is

```
DO variable = expr1 , expr2 [ , expr3 ] { command_list }
```

where the third expression is optional, defaulting to 1. The value of variable ($variable) is in turn set to expr1, expr1+exp3, ..., expr2, and the commands in command_list executed. Changing the value of $variable within the command list has no effect upon the loop. Do loops may be nested, but the name of the variable in each such loop must be distinct. A trivial example would be

```
DO val = 123, 123+10, 2 { WRITE STANDARD $val }
```

while a more interesting example would be the macro square discussed in the section on examples. Because the body of the loop must be scanned (and parsed) repeatedly, loops with many circuits are rather slow. If at all possible you should try to use vector operations rather than DO loops. For example the loop

```
DO i=0,DIMEN(x)-1 {
    SET x[$i]=SQRT(x[$i]) IF(x[$i] > 0)
    SET x[$i]=0 IF(x[$i] <= 0)
}
```

is better written as

```
SET x=(x > 0) ?  SQRT(x) :  0
```

where the ternary operator ?: is discussed in the section on vectors.

Foreach loops are similar, with syntax

```
FOREACH variable ( list ) { command_list }
```

where the list may consist of a number of words or numbers. Each element in the list is in turn defined to be the value of $variable, and then the commands in command_list are executed, so that for example the commands:

```
FOREACH i ( one 2 three ) { WRITE STANDARD $i }
```

will print out:

```
one
2
three
```

Foreach loops may be nested, but again the variables must be distinct. You can delimit the list with {} so that it can include keywords, but even then you can't have the word delete in the list of a foreach. Sorry.

If statements look similar, with syntax

```
IF ( expr ) { list } ELSE { list2 }
```

where the ELSE clause is optional, but if it is omitted the closing } *must* be followed by a newline (or explicit \n). See Appendix I if you want to know why.

If the (scalar) expression is true (*i.e.* non-zero), then the commands `list` are executed, otherwise `list2` is, if present.

Help

There is an online help command. Typing HELP <CR> or HELP HELP gives a list of the help menu, or HELP `keyword` gives help with that keyword. The help menu consists of any files in the directory specified by the entry `help` in the environment file, so for example HELP `data` types the file `data` in that directory. For all cases except HELP `help`, the file is filtered through a version of the Unix utility **more** which pages the file. When **more** offers you a '...' prompt, type ? to see your options. The same filter is used by e.g. LIST MACRO . If the command is HELP `word`, after HELP tries to print the file `word`, it looks to see if `word` is a macro, and if so prints its definition. If `word` is a variable, its value is then printed, and if `word` is also a vector HELP prints the dimension, followed by the help string associated with the vector `vector_name` (see section on vectors).

Save and Restore

If for some reason you want to stop a SM session for later resumption, and simply suspending the process, '^Z', is not sufficient, (for instance the machine is going down), then the SAVE command will write a file containing all your currently defined macros, variables, and vectors, along with your current history buffer as the macro `all`. You will be prompted before each class of objects is saved, or you can put the answers on the command line. The file is ascii, and can be edited if you so desire. The filename defaults to **sm.dmp** if not specified, or to the value of `save_file` from your **.sm** file. If some bug has crawled unbeknownst to us into SM, and results in some sort of panic (such as a segmentation violation), a save file called **panic.dmp** is written to your temporary directory, no questions asked.

To restart, RESTORE `filename` will read them all back, using the same default file as for SAVE if no filename is specified, and *replace* the current history buffer with the value of `all` from the savefile. Of course, you could write a macro to preserve the current buffer (see the definition of `edit_all` for hints). If the file wasn't written by SAVE it is assumed to be a SM history file, those written when you quit SM, and each line is assumed to be a command and written to the end of the history buffer. This is generally useful when you started SM in the wrong directory. It wouldn't be hard to write a macro to use RESTORE to read a history file into a macro.

One problem with SAVE is that it saves lots of macros, including some of the system ones. Specifically, all macros are saved except those beginning with "##". This can be avoided with the MACRO DELETE `filename` command, *e.g.* MACRO DELETE

`utils SAVE 1 1 1 MACRO READ utils`. The macro `sav` discussed under 'useful macros' will do this for you, and indeed not `SAVE` any macros that have been read with the `load` macro. This is probably the best way to use the `SAVE` command. In addition, `sav` also decides to save variables and macros, only prompting you about saving vectors. `sav` is a good candidate for overloading (as `save`), and indeed is one of the macros redefined by the `set_overload` command.

Vectors and Arithmetic

The basic unit of data in SM is the **vector**, a named set of one or more numbers. There is no limit to the number of vectors that may be defined. SM allows the user to read vectors from files or define them from the keyboard, to plot them and to perform arithmetic operations upon them. SM also supports string-valued vectors (where each element is a string), which are less useful, but can be used for such purposes as labelling points on a graph.

To read a vector `vec` from a column of numbers in a file, where the filename has been specified using `DATA` and, possibly, the range of lines in the file to be used has been specified using the `LINES` command, use `READ vec nn` where `nn` is the column number, or `READ { vec1 n1 vec2 n2 ... }` to read many vectors. It is also possible to read a row, using `READ ROW vec nn`, where nn is the row number in the file. See `READ` for how to read a string-valued vector. A vector may also be defined as `SET vec = x1,x2,dx` to take on the values `x1,x1+dx,...,x2`, where `dx` defaults to 1 if omitted. If a scalar is encountered where a vector is expected, it is promoted to be a vector of the appropriate dimension; all other dimension mismatches are errors. Example:

```
SET value = 5
SET x = 0, 50, 2
SET y = x
SET y = x*0 + value
SET y[0] = 2*pi
SET y = value
SET x=0,1,.1 SET y=IMAGE(x,1.5)
SET s=str
SET s='str'
SET s[0]=1.23
SET x=1.23
SET s=x
SET s=STRING(x)
```

will define a scalar, `value`, with a value of 5, then define a vector, `x`, with 26 elements, valued 0, 2, 4, 6,..., 50, then define another vector, `y` with size 26 and the same values as `x`, set all 26 elements of `y` to have the value 5, set the first element of `y` to be 2π, set `y` to be a vector with one element with value `5`, and finally set `y` to be a vector with the values taken from a horizontal cross-section from 0 to 1 through the current image. Unless a vector `str` is defined `SET s=str` is an error; to make `s` a string-valued vector use `SET s='str'`. `SET s[0]=1.23` makes `s[0]` have the value `"1.23"` (a string), as `s` is now a pre-existing string vector. An arithmetic vector `x` is then defined, and `s` is redefined as an arithmetic vector too – you must be careful when dealing with string vectors! Finally, we explicitly convert an arithmetic vector to a

strin-valued one with the STRING operator. This is a somewhat contrived example, designed mainly to illustrate the convenience of the SET command. The ability to set particular elements is mostly used in macros such as smirnov2 which calculates the Kolmogorov-Smirnov statistic from a pair of vectors.

If you don't have many data points, rather than type them into a file, and use READ vec nn to define a vector, you can use the command

```
SET vec = { list }
```

For example

```
SET r = 0,10
```

is equivalent to

```
SET r = { 0 1 2 3 4 5 6 8 9 10 }
```

In fact, { list } is an expression, so SET vec = 2*{1 3 1} is also legal. If the first element of a list is a word, the vector is taken to be string-valued: SET s={ William William Henry Steven } defines a 4-element string vector, or you can use a string in quotes: SET s=<'1' 2 3 4> (if you used SET s={'1' 2 3 4} the first element would be '1' rather than 1). Once a vector is defined, you can write it to a file for later study using the PRINT command.

A scalar may be an integer, a floating point number, a scalar expression, DIMEN(vector), or WORD[expr]. The last two are the dimension of a vector, and an element of the vector with expr a scalar subscript. *Note that subscripts start at 0 and that* [ ] **not** ( ) *are used to subscript variables.* The expression WORD[expr] is in fact allowed even if expr is not a scalar, in which case the result is a vector with the same dimension as expr, and with the values taken from WORD in the obvious way.

Once vectors are defined, they may be combined into expressions using the operators +, -, *, /, **, CONCAT and the functions COS(), SIN(), TAN(), ACOS(), ASIN(), ATAN(), ABS(), DIMEN(), INT(), LG(), EXP(), LN(), SQRT(), STRING, and SUM(). The meaning of most of these is obvious, DIMEN returns the number of elements in a vector, SUM gives the sum of all the elements, CONCAT concatenates two vectors, INT gives the integral part of a vector, and STRING converts a number to a string. The precedence and binding are as for C (or Fortran), and may be altered by using parentheses (CONCAT has a binding just below + and -). All of these operators work element by element, so

$$y = 2 + \sin(x)$$

is interpreted as

$$y_i = 2 + \sin(x_i)$$

If there is a size mismatch the operation will only be carried out up to the length of the shorter vector and a message is printed; if VERBOSE is 1 or more, the line where the error occurred will be printed too. The constant PI is predefined. In addition to these arithmetic operations, there are also logical operators == (equals), ! = (not equals), >, >=, <, <=, && (logical and), and || (logical or). The meanings of the symbols are the same as in C, and as in C the value 0 is false while all other values are true.

String vectors may only be concatenated, added, or tested for (in)equalility. Adding two string-valued vectors concatenates their elements, so

```
{ a b c } + { x y z }
```

results in the vector `ax by cz`.

There are also a number of less mathematical operations. If you have an `IMAGE` (*q.v.*) defined, you can extract a set of values using the expression `IMAGE(expr,expr)`, where the two `expr`s give the coordinates where the values are desired. Note that this may be used as a way of reading large data files that are stored unformatted. The expression `HISTOGRAM(expr : expr)` can be used to convert a vector into a histogram. The second expression is taken to be the centres of the desired bins: bin boundaries are taken at the mean points (and points in the first expression lying on a boundary are taken to fall into the lower bin. Note the use of ':' not ',').

Vectors may be assigned to, using the syntax

```
SET vec = expr
```

or

```
SET vec[ expr ] = expr
```

or

```
SET vec = WORD(expr)
```

or

```
SET DIMEN(vec) = number
```

or

```
SET vec = expr1 IF(expr2)
```

or

```
SET vec = expr1 ? expr2 : expr3
```

The first form sets `vec` to have the value `expr`, element by element, `vec` is the name of the new vector. The form `vec[expr]` may be used to assign to an element of a vector, as usual the index starts at 0. Before you can use `SET` to assign values to the elements of a vector, you must create it using one of the other forms of the `SET` command. `SET vec = WORD(expr)` uses the macro processor to allow a limited function call capability. `WORD` should be a macro taking one argument, which is the `expr` . When the macro finishes, the current value of the argument is the value of `vec`. An example may make this clearer. Suppose we define a macro `sq` with one argument as

```
SET $0 = $1**2
```

then the command

```
SET x = 0,7 SET y = sq(x)
```

is equivalent to `SET y = { 0 1 4 9 16 25 36 49 }` . If you want to define a vector to which you will subsequently assign values using `SET vec[ expr ] = expr`, you may use `SET DIMEN(vec) = number` which declares `vec` to be a vector of size `number`, and initialises it to zero. You can optionally supply a qualifier to the `number`, either a `.f` (the default), or a `.s` to specify that the vector is string valued.

If the `IF` clause is present, only those elements of `expr1` for which the corresponding element of `expr2` is true (non-zero) are copied into `vec`; in general `vec` will have a smaller dimension than `expr1`. The last `SET` statement (with ?:) again borrows from C. If $\texttt{expr1}_i$ is true, then $\texttt{vec}_i$ is set to $\texttt{expr2}_i$, otherwise it is set to $\texttt{expr3}_i$. In this form of conditional assignment, the dimension of `vec` is the same as that of the right hand side. It may look strange, but it can be just what you want.

Each vector also has a help field, which may be used to provide a string describing the vector. The field is set by

```
SET HELP vec str
```

and viewed by

```
HELP vec
```

If VERBOSE is one or more, if a vector is arithmetic or string will also be noted. Vectors may be printed using the

```
PRINT [ file ] [format] { vec1, ..., vecn }
```

command, where if the optional `file` is missing, the values are typed to the keyboard; if the optional `format` is omitted, a suitable one will be chosen for you. Any combination of string- and arithmetic-vectors may be printed. If a value exceeds 1e36, it is printed as a *. This is consistent with the convention used in reading data that a 'missing' number is represented as 1.001e36; see READ for details. Vectors may be deleted with the command

```
DELETE vec
```

and listed with the command

```
LIST SET
```

Vectors are listed in ascii order along with their dimension, and any help string specified using the SET HELP command

An IF clause has been added to the plotting commands, to allow only those elements of a vector which satisfy some condition to be plotted, for example

```
CONNECT x y IF(z>3*x)
```

Of course, you could have used the IF command and the regular CONNECT command if you had preferred.

Overloading

Sometimes you might wish that SM's authors had decided to make a command behave a bit differently, for instance that ERASE or QUIT wouldn't appear on the history list, or that SAVE deleted all the system macros before saving your environment. Of course, you can (usually) write macros to get around these annoyances, but you *can't* easily give them the same names as the original commands (for these examples, the macros are called `era`, `q`, and `sav`).

It is possible to change the meaning of keywords (to 'overload' them), but it can be confusing, primarily because your new commands may not behave the same way as this manual says they will. For example, if you were perverse, you could define `points` to mean QUIT. Another danger is that you could end up with a recursive call – for instance if you wrote your own version of `box` that did all sorts of cunning things, then drew a box. If you said `box` in your macro, then overloaded the keyword, you'd have a macro that called itself. If you tried to use it, nothing would happen for a while, and then you'd start getting messages about "extending i/o stack" until you hit ^C. Or if you redefined "help" to mean "DELETE HISTORY HELP" (in upper case to avoid recursive calls, and in case "delete" has been overloaded), then "set help vec Help string" won't work (you'd have to use "set HELP vec ...").

Despite these warnings, overloading the meaning of SM's keywords can be very convenient. There are two sets of system macros that do just this, the compatibility ones (see Appendix IX), and one called `set_overload` that is described below.

In addition to the semi-trivial use of overloading to allow you to type `erase` not `era`, it is possible to add extra functionality to simple commands. For example, `set_overload` defines `window` to save the window parameters in variables, and `box` then uses these values to label appropriate axes in touching boxes. Another example is that (when overloaded) `lines` saves the line numbers used, so that you can write a macro to print the top 10 lines of a file (it's called `head`).

So how do you do it? The command `OVERLOAD keyword #` will remove the special meaning of *lowercase* `keyword` if `#` is non-zero, or reinstate it otherwise. You can still use the uppercase form – you can't overload *that*. So now that *e.g.* `box` has no special meaning you can define it to be a macro. What the `set_overload` macro does is to define new meanings for a number of keywords, the new definitions are in the macro file **overload**. If you intend using them (and I do all the time) you should look at this file. You can get them loaded by default by having a line "overload" in your **.sm** file. If you don't like some, *e.g.* box, you can simply say `OVERLOAD box 0` in your private startup file (see 'private initialisation') which is run after the system startup.

Most of the changes are benign, but not all. For example, the new definition of `relocate` allows expressions, but it'll break if you try to say `relocate ( 100 1000 )` to move to absolute screen coordinates. You can still say `RELOCATE ( 100 1000)` of course, and that's why most of the system macros are actually written in uppercase. The definition of `box` (actually `bo`, which `box` calls) may seem very complex, but it has to deal with `box \n` as well as `box 1 2`, and it must know if you have used the `WINDOW` command. This brings up another point – if you overload keywords, you could slow SM down. It isn't that overloading is inefficient, it's just that the macros that replace the old keywords may do a good deal of work, `box` is a case in point. Even when the macro is short and to the point, it's still extra work to parse the original word and find its value as a macro.

Useful Macros

When you start SM the directory specified as `macro` in your **.sm** file is searched for a file **default**, and then the macro `startup` from that file is executed. At the time of writing of this manual, `startup` was defined as:

startup

```
## macro invoked upon startup
PROMPT :
FOREACH 1 { default_font device edit file_type macro macro2 \
        overload printer } {
    DEFINE $1 :
}
FOREACH 1 ( TeX_strings case_fold_search history_char \
        overload ) {
    DEFINE $1 :
    IF($?$1) {
        IF('$$1' == '0') {
            DEFINE $1 DELETE
        }
    }
}
IF($?device) { DEVICE $device
} ELSE { DEFINE device nodevice }
IF($?default_font $?TeX_strings == 0) {
    echo You can only define a default font if you use TeX
}
IF($?history_char) { # use $history_char as history character
    EDIT history_char $history_char
    EDIT ^ ^
    DEFINE history_char DELETE
}
# load the default mongo macros
DEFINE mfiles < stats utils > # used by 'sav'
FOREACH f ( mongo $mfiles ) { MACRO READ "$!macro"$f }
FOREACH var ( x_col y_col data_file ) { DEFINE $var . }
# load uppercase if defined in .sm file
IF($?uppercase) {
    MACRO READ "$!macro"uppercase
}
# and overload keywords such as erase, if so desired
set_overload $?overload
# and some keymaps
IF($?edit) { READ EDIT "$!edit" }
# and an optional macro file, with macro startup2
IF($?macro2) {
    MACRO READ "$!macro2"default
    DEFINE 1 (INT(WHATIS(startup2)/2))
    DEFINE 1 ($1 - 2*INT($1/2))
    IF($1) { startup2 } # startup2 is defined
```

```
}
# provide a \n after the IF
```

As this macro is executed every time that you run SM, let us consider it in some detail. After setting the prompt, it looks for entries for a number of variables in your **.sm** file. Some (such as `printer`) are simply `DEFINE`d, while some (such as `TeX_strings`) are only `DEFINE`d if they have a non-zero value. Because some of the values might not be numeric, the comparison is forced to be done on strings by enclosing the quantities in single quotes. If `device` is defined it is used to set the default plotting device, and both it and `printer` are used by a couple of macros (`hcopy` and `hmacro`) that produce hardcopy. The variables `TeX_strings` and `default_fonts` are used in producing labels; see Appendix X for details. Because TEX uses ^ for superscripts, we allow you to put a `history_char` line in **.sm** to specify a character to use rather than ^ for history (I use ‘). The variable `file_type` is used by the `IMAGE` command to determine the file format that you use (*e.g.* C, or unformatted fortran).

`Startup` doesn't have to check that `macro` was successfully defined as it must have been found for `startup` to have been read in the first place. `Macro` specifies where to look for macro libraries, and `startup` next sets the variable `mfiles` containing the names of some of the system macros to be loaded, and reads them. The macro `load` defined below also maintains the `mfiles` list, as does `unload`. It is used by the `sav` macro, which is discussed below the main listing of macros that follows. We also set some variables used by the `id` macro.

As part of our effort to be nice to users, if you have `uppercase 1` in your **.sm** file, we also load the `uppercase` macros. Next `startup` overloads some keywords if `overload` is in your **.sm** file, reads a file of keybindings (if `edit` is given in **.sm**), and finally tries to read a second optional macro directory `macro2`, and executes a macro `startup2` if it's defined (that's what the `WHATIS` is checking). This is quite important, as it provides a way to customise SM to your personal taste without convincing the local SM guru that your taste should be foisted on everyone. If you want a prompt that is different, or a definition of `q` that just quits without asking questions, you can get them by using `macro2`. You can see that it is possible to tailor SM pretty much as you wish without changing a line of code, just by playing with the `startup` macro.

SM provides various compatibility macros, and some to package often-used functions. The macro files **stats** and **utils**, which are read when SM is started, provide various useful macros, a few of which are presented here. To see a current list, either look at the files directly, set `VERBOSE` to zero and list all the macros, look at Appendix VIII, or use `lsm` to list macro files (this only works if you are running Unix). We give here a number of macros taken from the files **default**, **mongo**, **stats**, and **utils**. Among those not listed are those like `lin` defined to be `lines` that are pure abbreviations, those like `xlogarithm` defined as `SET x=lg(x)` which provide functionality in a perhaps familiar form, and many more like those that are given here which provide enhancements (*e.g.* the macro `barhist`). A discussion of a few of the more interesting or obscure follows. Keywords are written in uppercase, because you might have been playing tricks with overloading the lowercase equivalents.

Many of these macros, in fact all from **default** and **mongo**, start with ## so as not to show up in listings made when VERBOSE is 0, and so as not to be SAVEd. In the interest of brevity we have omitted most of these initial comments.

```
cumulate      2   # Find the cumulative distribution of $1 in $2
                  DEFINE sum 0 SET $2=0*$1 SET HELP $2 Cumulation of $1
                  DO i=0,DIMEN($1)-1 {
                      DEFINE sum ( $sum + $1[$i] )
                      SET $2[$i] = $sum
                  }
                  define sum delete
da            1   DATA "$1"
del1          1   DELETE HISTORY \n
dev           1   del1 DEFINE device $1 DEVICE $1
dra           2   # Draw, accepting expressions
                  define 1 ($1) define 2 ($2) draw $1 $2
edit_hist         # Edit the history list
                  del1 MACRO all 0 100000 # define "all" from buffer
                  WRITE STANDARD Editing History Buffer\n
                  MACRO EDIT all # do the editing
                  DELETE 0 100000 # empty history buffer
                  WRITE HISTORY all # replace history by "all"
era               del1 ERASE
gauss         1   # Evaluate a Gaussian :  N($mean,$sig)
                  SET $0 = 1/(SQRT(2*PI)*$sig)*EXP(-(($1-$mean)/$sig)**2/2)
get           2   # Syntax:  get i j.  Read a column from a file.
                  # Name of vector is jth word of line i.
                  DEFINE nn READ $1 $2 echo reading $nn\n
                  READ $nn $2
                  SET HELP $nn Column $2 from $data_file
                  DEFINE nn DELETE
hardcopy          DEVICE nodevice    # close old device
hcopy         13  ## hcopy [printer] [l1] [l2]:  Make hardcopy of playback buffer
                  # optionally specify which printer ($1) and which lines ($2-$3)
                  IF($?1) { DEVICE $printer $1 } ELSE { DEVICE $printer }
                  IF($?2 == 0) {
                      DEFINE 2 0 DEFINE 3 10000
                  } ELSE {
                      IF($?3 == 0) { DEFINE 3 $2 }
                  }
                  playback $2 $3 \nDEVICE $device
                  bell
hmacro        12  ## hmacro [macro] [printer] :  make hardcopy of 'macro' on 'printer'
                  # If only 1 argument is present, it is taken to be the printer
                  # If no macro is specified, make a temp one
                  del1 IF($?1) {
                 if($?2) { # 2 args
                 DEFINE _mac $1
                 DEVICE $printer $2
```

```
DEFINE _temp 0 # no temp macro
} ELSE { # 1 arg, take as printer
DEVICE $printer $1
DEFINE _temp 1 # need temp macro
}
 } ELSE { # no args
DEVICE $printer
DEFINE _temp 1 # need temp macro
 }
 IF($_temp) {
DEFINE _mac _mac
echo "Create temporary macro, exit with ^X"
MACRO EDIT $_mac
 }
 $_mac \n DEVICE $device
 IF($_temp) { MACRO $_mac DELETE }
 DEFINE _mac DELETE DEFINE _temp DELETE bell
```

load

```
# load macros in default directory
DEFINE macro :  # get default directory
MACRO READ "$!macro"$1 # read macro file
IF($?mfiles == 0) {
    DEFINE mfiles $1
} ELSE {
    DEFINE 3 0
    FOREACH 2 ( $mfiles ) {
        IF('$2' == '$1') { DEFINE 3 1 }
    }
    IF($3 == 0) { DEFINE mfiles < $mfiles $1 > }
}
```

load2 1

```
# load macros in (second) default directory
DEFINE macro2 :    # get directory
IF($?macro2) {
    MACRO READ "$!macro2"$1    # read macro file
} ELSE {
    echo Directory macro2 is not defined
}
```

logerr 3

```
# Syntax:  logerr x y error, where y is logged, and error isn't
SET _y = 10**$2
SET d_y = LG(_y + $3) - $2 ERRORBAR $1 $2 d_y 2
SET d_y = $2 - LG(_y - $3) ERRORBAR $1 $2 d_y 4
DELETE _y DELETE d_y
```

lsq 4

```
# Do a least squares fit to a set of vectors
# Syntax:  lsq x y x2 y2 Fit line y2=$a*x2+$b to x y
SET _n = DIMEN($1)    # number of points
SET _sx = SUM($1)    # Sigma x
SET _sy = SUM($2)    # Sigma y
SET _sxy = SUM($1*$2)    # Sigma xy
```

```
                    SET _sxx = SUM($1*$1)     # Sigma xx
                    DEFINE a ( (_n*_sxy - _sx*_sy)/(_n*_sxx - _sx*_sx) )
                    DEFINE b ( (_sy - $a*_sx)/_n )
                    SET $4=$a*$3+$b
                    FOREACH v ( _n _sx _sy _sxy _sxx ) { DELETE $v }
playback            ## define "all" from buffer, and run it
                    # with args, only playback those lines
                    IF($?1 == 0) {
                        DEFINE 1 0 DEFINE 2 10000
                    } ELSE {
                        IF($?2 == 0) { DEFINE 2 $1 }
                    }
                    del1 MACRO all $1 $2 all
read_old        1   del1 # read a Mongo file onto the history buffer
                    READ OLD temp $1
                    WRITE HISTORY temp MACRO temp { DELETE }
rel             2   # Relocate, accepting expressions
                    define 1 ($1) define 2 ($2) relocate $1 $2
reverse         1   # reverse the order of a vector
                    SET _i = DIMEN($1),1,-1 SORT < _i $1 > DELETE _i
sav             1   # Save to a file $1, don't save from files '$mfiles'
                    _save $1
_save           1   # Save to a file $1, don't save from files '$mfiles'
                    del1
                    FOREACH 2 ( $mfiles ) { MACRO DELETE "$!macro"$2 }
                    DEFINE 2 0 define 2 ?  { save vectors?  }
                    SAVE "$!1" 1 $2 1
                    FOREACH 2 ( $mfiles ) { MACRO READ "$!macro"$2 }
```

`Cumulate` is given as a way *not* to write macros if you can help it (in this case, I couldn't). A better example is `reverse` which reverses the order of the elements in a vector without resorting to a `DO` loop.

The macro `da` could have been defined to be `DATA`, but there are various special characters that appear in filenames, for instance try `data /usr/spool/junk`, or `data disk$data:[ETHELRED]junk.dat`. The macro `da` provides a set of double quotes to escape these unwanted interpretations. Incidently, `da "/usr/spool/junk"` won't work.

`DELETE HISTORY` deletes the last command on the history buffer, so `del1` alone on a line will delete itself, which can be used to prevent a command from appearing on the history list, for example changing devices with `dev`. `Dev` also defines a variable `device` which is used by the `hcopy` and `hmacro` macros to make hardcopies, while returning you to your initial device. The `startup` macro listed above also sets `device`, if it is specified in your **.sm** file. You should be careful *not* to include more than one `del1` macro in any macro that you write yourself, as each `del1` will remove a command from history and you could find commands mysteriously disappearing.

`Gauss` evaluates a Gaussian, *e.g.* `SET x=-3,3,0.05 SET g=gauss(x) lim x g box con x g`, an example of using a macro like a function definition. (For this example to work, you have to define variables `mean` and `sig` first).

There is an example of reading variables from files and using them in macro `get`. This reads a word from a line in a file with the `DEFINE nn READ i j` command, which sets `$nn` to be the j^{th} word on line `i` of the current data file. This variable is then used to `READ` a vector, which is given the appropriate name. So if a file looks like:

This is an example file

alpha	beta	gamma	delta
1	10	0.1	1e1
2	20	0.2	1e2
3	30	0.3	1e3
4	40	0.4	1e4
5	50	0.5	1e5

then the commands

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

will read '1 2 3 4 5' *into* vector `alpha`, '10 20 30 40 50' into `beta` and so forth. Note that

```
DEFINE READ file_id 1 LABEL $file_id
```

will write out 'This is an example file' to the current position of the plot pointer (see, *e.g.* `RELOCATE`). Incidently, `READ ROW omega 5` would set the vector `omega` to have values '3 30 0.3 1e3'.

The macros `hcopy` and `hmacro` make hardcopies of, respectively, the playback buffer and a macro. Both assume that the variables `device` and `printer` are set. `device` is set from your **.sm** file and by the `dev` macro; `printer` is assumed set in **.sm**. (See 'startup' file above). If all is well, the macros switch to device `printer` (with an argument to specify which sub-printer is desired. We have so many laser printers here...), execute the desired commands, and return to the initial `device`. When the `printer` device is closed, hardcopy will result. Note the use of `\n` to ensure that no nasty things happen; if there were no `\n` and the buffer ended with `LABEL Hi`, the plot could appear with a label `Hi device tek4010`. The versions of hcopy and hmacro given here accept a variable number of arguments ('13' means up to 3 arguments). The first (if present) is taken to be the desired laser printer, the next argument is the number of the first line that you want played back, and the third is the last line number. (If you omit both line numbers you'll get the whole buffer; if you omit the second you'll just get the one line). The macro sees what it has been given by using `$?` to see which variables are defined, and acts accordingly. `Hmacro` is somewhat similar, except that if you omit an argument it is taken to be the macro name, and a temporary one is created for you. The `playback` macro deals with its arguments in a similar way, and is discussed further in the examples at the end of this section.

`load` enables you to read a set of macros from a directory specified as `macro` in your environment file. `Load2` is similar, but it looks in directory `macro2`. The macro `unload` (not listed here) will undefine the `loaded` macros. Note that a list of all the `loaded` macros is kept in `$mlist`, which is used by the `sav` macro to avoid `SAVE`ing lots

of system macros. `Sav` is written in terms of a macro `_save` so that it won't itself be forgotten (by `MACRO DELETE`) while in the middle of saving macros.

If you want to put errorbars on logarithmic plots, `logerr` is the macro you've been looking for. It calculates the correct length for the errorbars, and plots them de-logging and re-logging as appropriate.

The macros `rel` and `dra` illustrate a method of using expressions, rather than numbers, in the commands `RELOCATE` and `DRAW`. See Appendix I if you want to know why `DRAW` won't accept an expression directly. These macros exploit the fact that the arguments to a macro are whitespace delimited, so a string such as `1+2/$x` comprises one argument. Redefining the arguments means that the macros don't have to define, and then delete, a couple of variables to hold the expressions.

Now that you have had your appetite whetted, we strongly recommend that you take the time to look through the other macros that are available, most of which should be listed in Appendix VIII. Otherwise how would you know that there are macros to draw arrows on plots, do KS and Wilcoxon tests on vectors, and a host of other good things?

More Examples of Macros

In all these examples, we'll use the `del1` macro discussed above to keep commands off the history list. Let's start with a Fourier series, to demonstrate SM's ability to manipulate vectors. All keywords are capitalised for clarity. Start SM, choose a plotting device (with the `dev` macro), and erase all the commands on the history (or playback) buffer with `DELETE 0 10000`. Then type the following commands:

```
SET px=-PI/10,2*PI,PI/200
SET y=SIN(px) + SIN(3*px)/3 + SIN(5*px)/5 + SIN(7*px)/7
SET y=(y>0)?y:0
LIMITS -1 7 y
BOX
CONNECT px y
```

The vector `px` could just as well have been read from a file. You should now have a part of a square-wave, truncated at 0.

Now consider a simpler way of doing the same thing. For the present, clear the history buffer again (`DELETE 0 10000`), and type:

```
SET px=-PI/10,2*PI,PI/200
SET y=SIN(px)
DO i=1,3 {
    SET val = 2*$i + 1
    SET y = y + SIN(val*px)/val
}
DELETE val
LIMITS -1 7 y
```

```
BOX
CONNECT px y
```

Here we use a vector `val` to save a value, an equivalent (but slower) loop using SM variables would be

```
DO i=1,3 {
    DEFINE val (2*$i + 1)
    DEFINE y = y + SIN($val*px)/$val
}
DEFINE val DELETE
```

That is all very well if you only ever wanted to sum the first four terms of the series. Fortunately there is a way to change this, using the macro editor. First define a macro consisting of all the commands on the history list:

```
del1 MACRO all 0 10000
```

will define the macro `all` to be history lines 0-10000. (You need the `del1` to avoid having the `MACRO all 0 10000` in your macro). Then you can edit it using

```
del1 MACRO EDIT all
```

when you have made the desired changes (e.g. changing `DO i=1,3` to `DO i=1,20`) use ^X to leave the editor and return to the command editor. Now you could type `all` to run your new macro, or put it back onto the history list. To do the latter, delete the commands now on the history list (the now-familiar `DELETE 0 10000`), then `del1 WRITE HISTORY all` to put the macro `all` onto the list. Now the `playback` command will run all those commands, and produce a better squarewave. (As discussed in a moment, `playback` is a macro so type it in lowercase, unless you have defined your own `PLAYBACK` macro.)

This ability to edit the history buffer is convenient, and there is a macro provided called `edit_hist` which goes through exactly the steps that we took you through. The trick of including a `del1` in macros is pretty common, for example `h` is defined as `del1 HELP` so that it won't appear on the history list. The macro `playback` is rather similar to `edit_hist`, but instead of editing and then writing `all`, it executes it. We discussed the possibility of just playing back a limited number of lines while talking about `hcopy`, just say `playback n1 n2`.

Now that you have a set of commands which produce a Fourier plot, it would be nice to define a macro to make plots, taking the number of terms as an argument, and then free the history buffer for other things. After a playback, the macro `all` is defined, so let's change its name to `square`. There is a macro `ed` defined more-or-less as `del1 MACRO EDIT`, so type `ed all` to enter the macro editor. Use ↑ or ^P to get to line 0 and change the number of arguments from 0 to 1, and the name of the macro from `all` to `square` (the space between the name and the : is required.) Then go to the `DO i=1,20` line, and change `20` to `$1`. Exit with ^X, clear the screen with `era` and type `square 10`. Now how do you save your nice macro? `WRITE MACRO square filename` will write it to file **filename**, and next time you run SM `MACRO READ filename` will define it. In fact there is a command `SAVE` to save everything which can be a mindless way of proceeding. A macro similar to this Fourier macro called `square` is in the file `demos` in the default macro directory (and was used to produce the top

left box of the cover to this manual). To try it yourself, type something like `load demos square 20`. (`20` is the number of terms to sum.)

If your wondering why `ed` is only 'more-or-less' defined as `del1 MACRO EDIT`, it's because the real `ed` checks to see if you have provided a macro name, and if you haven't it edits the same macro as last time.

But enough of macros which fiddle with the history buffer. Here are four sets of macros which do useful things, and may give some idea of the power available. First a macro to use the values of a third vector to mark points, then one to do least-squares fits to data points, then a macro to join pairs of points, and finally some macros to handle histograms and Gaussians. These macros are given in the format that SM would write them to disk (ready for a `MACRO READ`), with the name, then the number of arguments if greater than 0, then the body of the macro.

First the points.

```
alpha_poi    3  # alpha_poi x y z.  Like poi x y, with z as labels for points
                DO i=0,DIMEN($1)-1 {
                    DEFINE _x ($1[$i]) DEFINE _y ($2[$i])
                    RELOCATE $_x $_y
                    DEFINE _z ($3[$i])
                    PUTLABEL 5 $_z
                }
                FOREACH v (_x _y _z) { DEFINE $v DELETE }
```

Here we use the temporary variables `_x _y _z` to get around restrictions on expressions in `RELOCATE` commands. Note the `DO` loop running from `0` to `DIMEN($1)-1`, produced by array indices starting at 0 not 1. If you wanted to use character strings as labels, this could be done by using the `DEFINE READ` command, but this would be a good deal slower as SM would have to rescan the file for each data-point. The top right box of this manual's cover was made using this macro. The use of `alpha_poi` (and also the macro file called `ascii` has been superceded by the introduction of string-valued vectors into SM. Nowadays you'd simple read the column that you want to label the point with as a string (*e.g.* `READ lab 3.s`), set the point type to that string (*e.g.* `PTYPE lab`), and plot the points as usual (*e.g.* `POINTS x y`).

The least-squares macro makes heavy use of the `SUM` operator. It could be used to find the dimension of a vector too, but only clumsily, and `DIMEN` is provided anyway. The macro is:

```
lsq    4  # Do a least squares fit to a set of vectors
          # Syntax:  lsq x y x2 y2 Fit line y2=$a*x2+$b to x y
          DEFINE _n (DIMEN($1))              # number of points
          DEFINE _sx (SUM($1))               # Sigma x
          DEFINE _sy (SUM($2))               # Sigma y
          DEFINE _sxy (SUM($1*$2))           # Sigma xy
          DEFINE _sxx (SUM($1*$1))           # Sigma xx
          DEFINE a (($_n*$_sxy - $_sx*$_sy)/($_n*$_sxx - $_sx*$_sx))
          DEFINE b (($_sy - $a*$_sx)/$_n)
          SET $4=$a*$3+$b
          FOREACH v ( _n _sx _sy _sxy _sxx ) {DEFINE $v DELETE }
```

This does a linear fit, leaving the coefficients in `$a` and `$b`. It could be easily generalised to deal with weights, fits constrained to pass through the origin, quadratics...

Our third example connects pairs of points. This was written to deal with a set of data points before and after a certain correction had been applied.

```
pairs    4  # pairs x1 y1 x2 y2.  Connect (x1,y1) to (x2,y2)
            DO i=0,DIMEN($1)-1 {
                DEFINE _x ($1[$i]) DEFINE _y ($2[$i])
                RELOCATE $_x $_y
                DEFINE _x ($3[$i]) DEFINE _y ($4[$i])
                DRAW $_x $_y
            }
            FOREACH v ( _x _y ) { DEFINE $v DELETE }
```

After the introduction of vectors for `ANGLE` and `EXPAND` (in version 2.1) this macro can be rewritten to be much faster:

```
pairs    4  # pairs x1 y1 x2 y2.  Connect (x1,y1) to (x2,y2)
            DEFINE 6 { fx1 fx2 fy1 fy2 gx1 gx2 gy1 gy2 ptype angle expand aspect }
            FOREACH 5 { $!!6 } { DEFINE $5 | }
            ASPECT 1
            SET _dx$0=($3 - $1)*($gx2 - $gx1)/($fx2 - $fx1)
            SET _dy$0=($4 - $2)*($gy2 - $gy1)/($fy2 - $fy1)
            PTYPE 2 0
            SET _a$0=(_dx$0 == 0 ?  (_dy$0 > 0 ?  PI/2 :  -PI/2) :  \
                (_dy$0 > 0 ?  ATAN(_dy$0/_dx$0) :  ATAN(_dy$0/_dx$0) + PI))
            ANGLE 180/pi*_a$0
            EXPAND SQRT(_dx$0**2 + _dy$0**2)/(2*128)
            POINTS (($1 + $3)/2) (($2 + $4)/2)
            FOREACH 5 { angle aspect expand ptype } { $5 $$5 }
            FOREACH 5 { $!!6 } { DEFINE $5 DELETE }
            FOREACH 5 ( _a _dx _dy ) { DELETE $5$0 }
```

Note how `DEFINE name |` is used to save things like the angle and expansion, and that the name of the macro (`$0`) is used to make unique vector names, or at least names like `_dxpairs` that are very unlikely to be in use.

SM allows you to plot a pair of vectors as a histogram, but what if you have only got the raw data points, not yet binned together? Fortunately, SM can do this binning for you. Consider the following macro:

```
get_hist    6  # get_hist input output-x output-y base top width
               # given $1, get a histogram in $3, with the centres of the bins in $2
               # Bins go from $4 to $5, with width $6.
               set $2 = $4+$6/2,$5+$6/2,$6
               set help $2 X-vector for $3
               set $3=0*$2 set help $3 Histogram from $1, base $4 width $5
               do i=0,dimen($1)-1 {
                   define j ( ($1[$i] - $4)/$6 )
                   set $3[$j] = $3[$j] + 1
```

```
}
define j delete
```

(Since this was written, a new feature was added to SM, the expression `HISTOGRAM(x:y)`, to make histograms. The macro we discussed above can now be written much more efficiently as:

get_hist 6

```
# get_hist input output-x output-y base top width
# given $1, get a histogram in $3, with the centres of the bins in $2
# Bins go from $4 to $5, with width $6.
set $2 = $4+$6/2,$5+$6/2,$6
set help $2 X-vector for $3
set $3=histogram($1:$3)
set help $3 Histogram from $1, base $4 width $5
```

)

Suppose that your data is in vector x, for want of a better name, and it has values between 0 and 20. Then the command

```
get_hist x xx yy 0 20 1
```

will produce a histogram in `yy`, bin centres in `xx`, running from 0 to 20 with bins 1 unit wide. So you could plot it with `lim xx yy box hi xx yy` , and maybe it looks like a Gaussian. So what is the mean and standard deviation? The command

```
stats x mean sig kurt echo $mean $sig $kurt
```

will answer that, and find the kurtosis too. (Macro `stats` consists of lines such as `define $2 ( sum($1)/dimen($1) )`). Then we could use the macro `gauss` to plot the corresponding Gaussian,

```
set z=0,20,.1 set gg=gauss(z) set gg=gg*dimen(x) con z gg
```

The bottom left box of the cover was made this way. What if you don't like the way that the histogram looks? Try the macro `barhist`. Now, if you wanted to plot a lognormal, you'd have to write your own macro, and you could use `SORT` to find medians and add another macro to `utils`, followed by one to find Wilcoxon statistics... (Since this was written a Wilcoxon macro was donated to `stats`).

Keywords are reserved, so don't try to use them as macro names or whatever. You can use the lowercase forms if you explicitly ask to be allowed to overload them. The control words are :

```
WORD STRING FLOAT INTEGER ( ) { } \n ! CHDIR DEFINE DELETE DO EDIT ELSE FOREACH HELP
HISTORY IF KEY LIST MACRO OLD OVERLOAD PRINT PROMPT QUIT READ RESTORE RETURN ROW SAVE
SET STANDARD TERMTYPE USER VERBOSE VERSION WRITE
```

Arithmetic words are :

```
ABS ACOS ASIN ATAN CONCAT COS DIMEN EXP FFT INT LG LN PI POWER_SYMBOL SIN SQRT SUM TAN
WHATIS [ ] = + - * / < > | & ? :
```

SM plotting keywords are

```
ANGLE APROPOS ASPECT AXIS BOX CONNECT CONTOUR CTYPE CURSOR DATA DEVICE DOT DRAW ERASE ER-
RORBAR EXPAND FORMAT GRID HISTOGRAM IMAGE LABEL LEVELS LIMITS LINES LOCATION LTYPE LWEIGHT
MINMAX NOTATION POINTS PTYPE PUTLABEL RANGE RELOCATE SHADE SORT SPLINE TICKSIZE WHATIS
WINDOW XLABEL YLABEL
```

These are described below, with the convention that arguments between square brackets are optional. This has nothing to do with their use in subscripting arrays!

Glossary

.sm See Environment File.

Environment File When you start SM it looks in a file (by default called **.sm**) to discover where to find files that it needs (such as the default macros, the help files, and the font files). You can access variables stored in the environment file yourself, although this is probably seldom done by non-gurus.

Expression An SM expression is something that can appear on the right hand side of a `SET` command. More specifically, it is something that can appear as *part* of the right hand side of a `SET` command (this excludes implied do loops: `SET x=1,10,2`). Expressions may also appear in other contexts, such as in the `ANGLE` command, or in `DEFINE name ( expression )`. A formal definition of an expression is given by the YACC grammar given in Appendix IV, as the non-terminal symbol **expr**.

Filecap Binary files produced by different programmes (and languages, and even compilers) are not identical, Fortran 'unformatted' files being a glaring example. A filecap file is a database used to describe the byte-by-byte format of binary files, to allow SM to read them (using the `IMAGE` command).

Graphcap There are a very large range of graphics terminals (and laser printers and so forth) in this world, and each seems to have its own set of commands. A graphcap file is a database that is able to describe (almost) all of these dialects, allowing SM to plot on a wide range of devices

History SM remembers commands as you type them, so that you can repeat them or modify them (which includes correcting mistakes). The set of remembered commands is referred to as the history buffer.

List The word `list` is used in a few places in the manual in the specific sense defined by the YACC grammar in Appendix IV. A list is simply a list of words or numbers, and its meaning depends on the context. For example, `SET x=3*{1 2 3}` will set the vector `x` to be `2 4 6`, while `MACRO hi { echo Hello}` will define the macro `hi`.

Macro A macro is an abbreviation for a set of commands, so instead of typing a complicated sequence of commands you can simply type the macro's name. You can either think of macros as a new commands in their own right or as subroutines.

Mongo Mongo is a plotting package written by John Tonry, and widely used in astronomy departments. SM's command language was based on Mongo's, and we have provided some support for an easy transition from Mongo to SM.

Stdgraph Stdgraph is SM's device driver that uses graphcap (*q.v.*).

Termcap Terminals come in many, many, flavours and types. Their peculiarities are described by a termcap file, allowing SM's command editor to run on (almost) any terminal.

TEX TEX is a typesetting language developed by Donald Knuth. We provide an emulation of certain parts of TeX's mathematics mode in SM's label commands.

Overload A keyword (such as **DEFINE** or **SET**) is said to be *overload*ed if its meaning has been changed. Usually this will be by adding functionality, rather than by actually changing what it does.

Variable A variable is an abbreviation for a string, and may appear anywhere that the string in question could appear. Even if the variable contains a number (*e.g.* `6.62559e-34`) it is still just a string, although SM may choose to treat the string as a number in some contexts (*e.g.* the right-hand side of a **SET** command).

Vector A set of one (actually, zero) or more elements. The elements can be either numerical (floating point) or strings. Vectors are SM's primary data type. Do not confuse a 1-element vector (a scalar) with a variable (*q.v.*).

YACC The SM command language is written in a language called YACC (which is supported on Unix systems). We have provided an implementation of YACC called Bison in the SM distribution.

Syntax: ANGLE `expr`

For most purposes, only the first element of the expr is used, let's call it `D`, as it's an angle in degrees.

ANGLE will cause text from LABEL to come out `D` degrees anticlockwise from horizontal. It also causes points to be rotated counter-clockwise by `D` degrees.

If `D` is non-zero it will force axis and other labels to be written with SM's internal fonts, and will overrule the tendency to put x-axis labels horizontal, and y-axis labels vertical.

For plotting points the full vector of values is used, with the point rotated by the value of expr. If more points are specified than the dimension of expr, the first element will be used for the excess.

Syntax: APROPOS `pattern`

Apropos lists all macros whose name or introductory comments match the given pattern. Probably the most common use for the command is simply to look for a word – *e.g.* `APROPOS histogram`.

The pattern is a slightly restricted version of a normal Unix regular expression, specifically:

`.`	Matches any character except \n
`[...]`	Matches any of the characters listed in the `[ ]`.
	If the first character is a ^ it means use anything except the range specified (in any other position ^ isn't special)
	A range may be specified with a - (e.g. [0-9]), and if a] is to be part of the range, it must appear first (or after the leading ^, if specified). A - may appear as the special range ---.
`^`	Matches only at the start of the string. Note that you must quote the ^ with an `ESQ-q` if you use it as your history character – look in tehe index under `history` to learn how to change it.
`$`	Matches only at the end of a string
`\t`	Tab
`\n`	Newline
`\.`	(any other character) simply escapes the character (*e.g.* \^)
`*`	Any number of the preceeding character (or range)
`?`	One or more of the preceeding character or range

By default searches are case sensitive, but you can make searching ignore case by defining the variable case_fold_search (you can do this by putting a line `case_fold_search 1` in your **.sm** file if you so desire).

The name and the comments are searched separately, so you could list all macros beginning with a, b, c, d, e, or z by saying

```
APROPOS ^[a-ez]
```

(which works because all comments start with a #), or

```
APROPOS "^#[^#]"
```

to list all macros that start with a single # (the quotes are needed to stop the #'s from being treated as comment characters).

Syntax: ASPECT A

Set the aspect ratio (Y/X) to be A; This is used in drawing characters and points. If A is exactly 0, the current aspect ratio is printed (you can also calculate this is yr/xr, where xr and yr are specified in the graphcap entry for the device, *e.g.* `:xr#1024:yr#780:`. It is reset when a new DEVICE command is issued.

Usually the aspect ratio is calculated by SM to make characters look right (and to make square points square), but it is sometimes useful to override this, especially when positioning labels on graphs that will be plotted on printers of different aspect ratios.

Arithmetic

Arithmetic is allowed on vectors and scalars in SM, using the following operators, where expr is a expression and vector the name of a vector.

Nonary: (?)

PI	Pi

Unary:

-expr	Change sign	ABS(expr)	Absolute value
ACOS(expr)	Arccosine	ASIN(expr)	Arcsine
ATAN(expr)	Arctangent	COS(expr)	Cosine
DIMEN(vector)	Dimension of a vector	EXP(expr)	Exponential
INT(expr)	Integral part	LG(expr)	Log_{10}
LN(expr)	Log_e	SIN(expr)	Sine
SQRT(expr)	Square root	STRING(expr)	Convert to a string
SUM(expr)	$\sum_i \text{expr}_i$	TAN(expr)	Tangent
VECTOR[expr]	Elements of an array	(expr)	Raise precedence

Binary:

expr + expr	Add	expr - expr	Subtract
expr CONCAT expr	Concatenate	expr * expr	Multiply
expr / expr	Divide	expr ** expr	Exponentiate

There are also some special operators:

HISTOGRAM(expr:expr)	Construct histogram
IMAGE(expr,expr)	Extract cross section
(expr1 ? expr2 : expr3)	expr2 if expr1 is true, else expr3 + multis

(The form with `?:` is similar to the corresponding SET command, but it needs parentheses if used as an expression). The expression `VECTOR[expr]` results in a vector of the same dimension as the `expr`, with elements taken from `VECTOR` (*i.e.* `VECTOR[INT(expr`$_i$`)]`). See, for example, the macro `interp`.

The precedences are what you'd expect, with ** being highest, then * and /, then + and −, and then CONCAT. The logical operators all have even lower precedence than CONCAT.

If you have defined an image file with the IMAGE command, `IMAGE(expr,expr)` is an expression to extract values from your image. The two expressions give the x and y values where the image is to be sampled. For example `SET x=0,1,.01 SET z=IMAGE(x,0.5)` will extract a horizontal cross section through an image.

`HISTOGRAM(expr1:expr2)` constructs a histogram from a vector, where the data is in `expr1`, and `expr2` (which must be sorted) gives the centres of the bins. Values on bin boundaries go into the higher bin up.

See 'Logical' for the logical operators, and 'whatis' for finding out if strings are numbers, words, vectors, or whatever.

Syntax: AXIS A1 A2 ASMALL ABIG AX AY ALEN ILABEL ICLOCK

Makes an axis labeled from A1 to A2 at location AX, AY, length ALEN. If ABIG > 0 use that for spacing of large ticks. If ASMALL < 0 make a logarithmic axis, if ASMALL = 0, do the default. (See TICKSIZE for more on the meaning of negative ASMALL and/or ABIG). If ASMALL > 0 try to use that for the spacing of small ticks. ILABEL is 0 for no labels, 1 for labels parallel to axis, 2 for perpendicular to axis, and 3 for neither labels nor ticks. ANGLE determines the angle of the axis. ICLOCK = 1 for clockwise ticks on the axis, 0 for anticlockwise.

For example, if the limits were 0 1 0 1, then the following commands would be equivalent to BOX:

```
AXIS 0 1 0.05 0.2 3500 3500 27500 1 0
AXIS 0 1 0.05 0.2 3500 31000 27500 0 1
ANGLE 90
AXIS 0 1 0.05 0.2 3500 3500 27500 2 1
AXIS 0 1 0.05 0.2 31000 3500 27500 0 0
ANGLE 0
```

(If expand is 1, that is).

Rather than use AXIS to draw all of your axes, it may be easier to use BOX with some 3's to disable its axis-drawing habits. You'll still get a box, but no ticks or labels.

This has changed in V2.1: To specify logarithmic axes you should now specify the logarithms, just as you do to box. For example, to draw a logarithmic axis running from 1 to 1000, specify A1 as 0 and A2 as 3, rather than 1 and 1000.

Syntax: BOX [INTEGER1 INTEGER2 [INTEGER3 INTEGER4]]

BOX puts axes around the plot region, labelling the lower and left according to the values set by LIMITS and TICKSIZE. If arguments INTEGER1 and INTEGER2 are included (default 1 and 2) they are used as ILABEL arguments for the lower and left axes (see AXIS). An ILABEL of 0 means to omit axis labels, 1 produces labels parallel to the axis, 2 perpendicular, and 3 omits both labels and tickmarks. INTEGER3 and INTEGER4 are used for the top and right axes.

If you want to change the font used for axis labels, define the variable default_font, either interactively (DEFINE default_font oe), or by putting a line in your **.sm file**: default_font oe. This affects regular as well as axis labels, and only works if you use TeX_strings, which we recommend anyway.

Syntax: CHDIR WORD

Set the current directory to be WORD, where WORD is any valid directory. It might be wise to enclose it in quotes, *e.g.* `CHDIR ''[-.more_data]"`, or use the `cd` macro. The new directory only affects SM, for example DATA or SAVE commands. When you exit SM, you will be back where you started. If the directory starts with a '~', the '~' will be replaced by the name of your home directory. This is the only place that '~' is significant; in particular it will not be recognised by the DATA command. *

Syntax: CONNECT WORD1 WORD2 [IF (expr)]

CONNECT draws line segments connecting the points in vectors WORD1 and WORD2. If the IF clause is present, only connect those points for which expr (see the section on vector arithmetic) is non-zero.

In fact, either or both of the WORDs may be replaced by 'parenthesised expressions', *i.e.* expressions in parentheses. For example,

```
CONNECT x (2*y)
```

plots x against 2y.

If WORD1 and WORD2 have different dimensions CONNECT will ignore the excess points in the longer vector. If you want to plot a constant value you'll have to explicitly promote it, for example

```
CONNECT x (1+0*x)
```

which makes a rather boring plot.

To draw a line in a label you can either use CONNECT or DRAW, or use the TEX-macro `\line` to directly insert your line.

Syntax: CONTOUR

Makes a contour plot of an image read by the IMAGE command (*q.v.*). The contour levels are set using the LEVELS command. Contrary to previous versions of the CONTOUR command, plot coordinates are taken to be those set by the LIMITS, and contours are drawn in the current LTYPE. It is not possible to produce labeled contours.

See also the IMAGE CURSOR command for using cursors to get values from images, MINMAX for finding the minimum and maximum of images, and Arithmetic for extracting cross-sections of images.

* Due to a VMS RTL bug, this command is not available on all VMS systems.

Syntax: CTYPE WORD
CTYPE INTEGER
CTYPE = expr

With WORD, set the line colour to be WORD, if your display device supports coloured lines, where WORD must be one of default, white, black, red, green, blue, cyan, magenta, or yellow. The colours are those composed of three, zero, one, or two of the primary colours red, green, and blue. When a device is opened it sets default to some device specific value (*e.g.* white for xwindows, black for sunwindows).

Initially, CTYPE INTEGER is another way of selecting the same colours as are available with CTYPE WORD, where CTYPE 1 is the equivalent of the first colour listed above, or white (so default is 0). However, the CTYPE = expr command redefines the available colours to be the elements of the array given by expr. Each element of this array is interpreted as RED + 256*GREEN + 256^2*BLUE for the given colour, where 0 is off, and 255 corresponds to full intensity. So another way to get white lines would be to say:

```
CTYPE = { 0 255 0 } + 256*({ 0 255 255 } + 256*{ 0 255 0 })
CTYPE 1
```

while

```
CTYPE 2
```

would give green lines. You can reset the colours to their default (*i.e.* correct) values using the macro reset_ctype.

Many devices (*e.g.* suns) require you to specify a number of colours that is a power of 2, so asking for 70 colours will use up 128 slots. It is probably a good idea to use as few colours as possible, as they are scarce resources on most displays. You should also be aware that the display may use some of 'your' slots for the background, so specifying 63 colours on (*e.g.*) a sun actually requires 64 (and asking for 64 will use up 128). If you specify more colours than are physically available, or more than the device driver thinks that you deserve, SM will interpolate your values of CTYPE for you.

The default colour is specified in the device drivers, or in the DC (Default Colour) graphcap capability, and is set whenever a device is opened, so don't try to modify it with a CTYPE = expr command. You can, however, override the default colour with a foreground entry in your **.sm** file; it should be the name of a colour (as listed above). You may also be able to specify a background colour (as background). This is either a colour name or a set of three integers in the range 0-255 specifying the red, green, and blue values. We allow you this chance to specify arbitrary colours because it's your only chance to affect the background, and you can't use a CTYPE = command to compose your own palette. On some devices the name of the background colour may be chosen from a wider selection; for example if you are using Xwindows you may use any name from the colour database.

Syntax: CURSOR
CURSOR `WORD1 WORD2`

Display the crosshair cursor to enable you to get positions (in user coordinates). Positions are typed on the screen. Exit the cursor routine by hitting the 'e' or 'q' key. If you exit with 'e', CURSOR issues a relocate command to set the current plot position to the cursor position, and puts the command in the history buffer. If you exit with 'q', no entry is made in the buffer. Usually successive positions overwrite each other, but if a digit is used to mark a point, then the next position noted will appear on the next line. (You can remember that digits lead to numerous values appearing).

Many graphics devices have things called "GIN terminators". SM usually expects that this be set to 'Carriage Return' with no extra characters, EOT is a popular (unacceptable) choice. If you have trouble check your graphics setup screen, then with your SM Guru who can look up in graphcap to see what is expected. If the local Guru were very friendly, he could change your GIN terminator to anything he wanted, even EOT, but he probably isn't.

The other form enables you to define a pair of vectors `WORD1` and `WORD2`. SM provides you with a cursor, and every time that you hit a key (except 'e'), it draws a point of the current type at the current position, and enters the (x,y) coordinates in the vectors. Exit with 'e', or abort with a ^C in which case `WORD1` and `WORD2` are unchanged.

Note that if you want to use SPLINE on the vectors produced in this way, you should take care that at least one of the vectors is monotonic and increasing, or use the SORT command.

See also IMAGE CURSOR which returns the value under the cursor as well as the position if an IMAGE (*q.v.*) has been defined.

For devices with mice, if the buttons do anything, they should generate the characters 'e', 'p', and 'q' (starting at the left).

The SunWindows cursor is slightly different. The cursor position is given by a pointing finger (it's the best we could do), and SM won't see any characters typed at the keyboard until you hit a carriage return. Device `sunwindows` is obsolete anyway, you should simply switch to using the `sunview` driver. *Its* cursor has a bug, in that it only sees every other character typed at the keyboard. If I knew why I'd fix it.

Syntax: DATA `file`

Use file `file` as the source of data read with the READ command. The file is assumed to have numerical data in columns separated by spaces, or tabs. The range of lines specified by LINES is reset. If the file can't be opened for read, you will be warned. The variable `$data_file` is set to `file`. See the READ command to see how to read the data. You may need to quote the filename, *e.g.* `DATA "/usr/file"`, which you can do by using the macro `da`: `da /usr/file`.

Syntax: DEFINE `name value`
DEFINE `name { value_list }`
DEFINE `name < value_list >`
DEFINE `name` DELETE
DEFINE `name ( expr )`
DEFINE `name :`
DEFINE `name |`
DEFINE `name ? [ { prompt } ]`
DEFINE `name ? [ < prompt > ]`
DEFINE `name` READ `INTEGER`
DEFINE `name` READ `INTEGER INTEGER2`
DEFINE `name` IMAGE
LIST DEFINE `[ begin end ]`

All of these varieties of DEFINE define variable `name` to have some value, but as variables can be defined in all sorts of ways there are a good many possibilities.

`Name` is a single word starting with a letter, and containing only letters, digits, or '_', and may be a keyword. Whenever SM comes across `$name`, it is interpreted as a reference to variable `name` and `$name` is replaced by its value. (Note that the variable 'date' is special, as it always contains the current date and time, try `echo $date` sometime.)

If you just want to know if a variable is defined, then `$?name` is defined to have the value 1 if `name` is defined, and 0 otherwise. Variables are not usually expanded within double quotes or { } , but if you use the syntax `$!name` the variable will be expanded within double quotes; `$!!name` will be expanded anywhere.

For the variants of DEFINE `name value` and DEFINE `name value_list`, `value` is either a word or number, or a list. The difference between using { } and <> to delimit a list is that keywords can appear within { } , but variables are not usually expanded.

DEFINE **name** DELETE, deletes a variable (see also the macro undef to undefine variables).

DEFINE **name** **(** **expr** **)** defines a variable to have the value of a (scalar) expression. When possible, it is more efficient to use vectors to perform calculations on scalars, rather than putting them into variables. It is also more efficient (and more obscure!) to use numbered variables (macro arguments) than real named ones. As a special dispensation, the expression can be an element of a string-valued vector (elements of arithmetic vectors are allowed too of course).

DEFINE **name** : defines the variable **name** from the environment file. **DEFINE name |** is used to define a variable from an internal SM variable such as **expand** or **angle**. The variables accessible are **angle**, **aspect**, **ctype**, **expand**, **fx1**, **fx2**, **fy1**, **fy2**, **gx1**, **gx2**, **gy1**, **gy2**, **ltype**, **lweight**, **ptype**, **uxp**, **uyp**, **verbose**, **xp**, and **yp**. The current plot limits are **fx1** etc., (or **gx1** etc. in device coordinates), the current position (in user coordinates) is **(uxp,uyp)** , the current position (in plot coordinates) is **(xp,yp)** and the rest should be obvious.

DEFINE **name** **?** will prompt you for the value of **name** at the keyboard, using the prompt string if given, otherwise the name of the variable. The old value of the variable (if defined) is printed within [], and is taken to be the default if you simply hit carriage return. As previously discussed, the difference between { } and <> is in the treatment of keywords and variables. If you don't want to use { } (probably because of something weird to do with when variables are expanded), you can always use quotes within <>.

Examples

```
DEFINE v1 5.993
DEFINE label1 KPNO
DEFINE label1 < National Optical Astronomical Observatory >
DEFINE v2 ($v1 + 3.4)
DEFINE v1 DELETE
DEFINE age ?  { How old are you?  }
DEFINE macros :  WRITE STANDARD "$!macros"
```

(Note that we couldn't have used <> to prompt for your age, because then the ? after **you** would be treated as a keyword).

The versions of the DEFINE command including READ define variables from the current data file. DEFINE **name** READ **INTEGER** sets **name** to be line **INTEGER** of the current data file, while DEFINE **name** READ **INTEGER INTEGER2** defines **name** to be word **INTEGER2** of line **INTEGER**. **name** is subject to the usual restrictions. For example, given a file with the following lines:

```
This is a file containing astronomical data
Magnitude Intensity Wavelength Error
```

Then using the **DEFINE** commands as follows:

```
DEFINE title READ 1
DEFINE labelx READ 3 2
```

will assign the string `This is a file containing astronomical data` to the variable `title`, and the word `Wavelength` to the variable `labelx`, so you can say `XLABEL $labelx`.

DEFINE `name` IMAGE defines a variable from a file read with the IMAGE command. Currently this only works for `X0`, `X1`, `Y0`, `Y1`, or any keyword from a FITS header.

LIST DEFINE lists all currently defined variables, or all those which are between `begin` and `end` alphabetically (asciily).

Syntax: DELETE [`INTEGER1` [`INTEGER2`]]
DELETE HISTORY [`INTEGER1` [`INTEGER2`]]
DELETE `WORD`

Delete commands `INTEGER1` to `INTEGER2` (inclusive) from the history buffer. If the `INTEGER`s are not present, delete the last command. DELETE 0 will delete all history commands.

The DELETE HISTORY commands are identical to the DELETE commands, except they themselves *do* appear on the history list; they are preserved for backwards compatibility and because DELETE HISTORY \n can be used to prevent a command from appearing on the history list (the macro `del1`).

DELETE WORD deletes the vector `WORD` (see SET `WORD` or READ `WORD` for how to define vectors.)

Syntax: DEVICE `WORD [ rest_of_line]`
DEVICE `INTEGER [ WORD ] [ rest_of_line]`

Choose a device to plot to. There are currently 9 devices available, `nodevice` (0), `stdgraph` (1), `raster` (2), `imagen`, `sunwindows`, `sunview`, `xwindows`, `x11`, and `upc` (a Unix PC). The first three are always available, with numbers as given, while the existence and numbers of the others depends on how SM was compiled. You never need refer to them by number anyway. When using `raster` you'll probably have to specify another device name as well, *e.g.* `DEVICE raster laserjet`, or (more likely) say `DEVICE laserjet` and put an `RA` in `laserjet`'s graphcap entry (*q.v.*). The X-windows and SunView drivers will create a plot window for you, while the Sunwindows should be run from inside a graphics tool.

Fortunately, stdgraph is capable of driving almost all graphics terminals in use, and the `WORD` argument specifies which one you want. If you omit `INTEGER`, and the device is otherwise unknown, it is assumed to be `stdgraph`. Stdgraph works by using a file called graphcap to tell it about graphics capabilities, see Appendix II for details. Anything else on the line is passed to the device driver.

Some examples may clarify this a little. DEVICE `nodevice` and DEVICE 0 are equivalent, and specify the null device, which does nothing. This is the default. DEVICE `1` `tek4010`, DEVICE `stdgraph` `tek4010`, and DEVICE `tek4010` are all equivalent, and specify that the device is to be `stdgraph`, using the graphcap entry for a Tektronix 4010 terminal.

Among the devices supported by stdgraph are `tek4010`, `tek4012`, `pericom`, `selanar` (or `hirez`) , `versaterm` (or `macvt`) , `vt640` (a vt100 with retrographics) , `vt240` in REGIS mode, `hard4012` or `hard4010` for a tek4010 that really has no decent ascii mode, `tek4025`, `wyse1575`, `cit414a` or 414a. We also support graphcap drivers for Postscript, QMS, and LN03 laser printers, (device names `postscript`, `qms` and `ln03`). Stdgraph can also coöperate with `raster` devices, for instance to plot on a lineprinter. There are probably other devices not listed here, added to graphcap since this was written.

Especially for hardcopy devices, you may have to specify which one you want, *e.g.* DEVICE `postscript` `latypus`. Because this depends on how your local graphcap was configured you'll have to see your Guru for guidance.

When a device is opened, it is set to the current CTYPE, LWEIGHT, and LTYPE, and the proper aspect ratio is chosen to make text and plotted points look nice. It also looks for an entry `foreground` in your **.sm** file, and uses it as the default colour for the device (this overrides any default that the device driver may have specified). The device driver may (or may not) choose to honour a `background` entry as well. These colours may be specified either as names (see CTYPE), or the background colour may be given as a set of three numbers, which are interpreted as the red, green, and blue intensities in the range 0 - 255. Some devices may allow you a wider selection of background names; for example the Xwindows driver allows any name from the colour database.

Use the LIST DEVICE command to see which devices are available, either in graphcap or as hard-coded drivers. Different ways of plotting to the same device (*e.g.* portrait or landscape) are accomodated by using different drivers (*e.g.* `postport` and `postland` for postscript devices) rather than some magic command to SM.

For the `stdgraph` (*i.e.* default) device driver the final word on the commad line (if present) is taken to be the name of a file to recieve the output that would ordinarily go to the screen, so if you say

```
device graphon outfile
```

and then create a plot nothing will seem to happen. However, if you close the device and write **outfile** to the terminal (maybe using /passall if you are running VMS) your plot will appear. In addition, any word beginning with a colon will be taken to be part of a graphcap entry (see Appendix II), and prepended to the entry in the graphcap file for your chosen device. For example, if you wanted to save your postscript output in a file you could say

```
dev postscript ":SY=echo File is $F:"
```

which would replace the `SY` entry that sent the output to the printer by a new one that merely tells you what the file is called. If you'd prefer to give it a memorable

name, you could say

```
dev postscript ":SY=echo File is \$F:OF=name:"
```

or

```
dev postscript :SY@:  :OF@:  name
```

(it doesn't matter if the entries are all in the same word). The former redefines the output file `OF` to be "name", and makes `SY` tell you so. The latter disables both `OF` and `SY`, so the generated postscript would ordinarily go to the terminal (just like any other graphics terminal), but a file **name** is specified, so the output is sent there instead.

As the `sunwindows` driver is now obsolete, and may well disappear in some future release, you should use the `sunview` driver instead. If you insist on using the old driver, if must be run from within a gfxtool.

The SunView window driver supports a subset of the usual SunView command line arguments, specifically:

-WH	Summarise options
-Wi	Open window as an icon
-Wl label	Specify label for the window (default: SM)
-Wn	Don't label window
-WP x y	Position of icon
-Wp x y	Position of window
-Ws w h	Size of window

The standard SunView popup 'frame' menu has been modified to allow you to erase the graphics screen. It is perfectly safe to use the menu to `quit` the graphics window, in this case the next `device sunview` command will create a new one. If SM thinks that the window is active when you try to kill it it will warn you; failing to believe it may result in a cascade of complaints to the console window.

There are two X-Windows drivers, one for X10 and one for X11 and they differ in their treatement of command line arguments. THe X11 driver is considerably more sophisticated.

The X10 driver is known as `xwindow`, and you can optionally specify a device to open, and a window ID, on the command line. For example

```
device xwindows DEVICE unix:0
```

will open a graphics window on device `unix:0`. (You can optionally include a `ID number` after the DEVICE if you are calling SM from a programme, and have already opened the window). The X10 driver doesn't bother to remember any hardware characters that you may have written on a graph, so that if you refresh the window they won't appear. If this worries you can, as always, force the software character set with an `expand 1.001`. Redrawing only occurs when SM is thinking about graphics, so a damaged window can stay on the screen for a while. A simple way to refresh it is to reopen the device – `device xwindows`.

The X11 window driver (device `x11`) supports a subset of the standard command line arguments, specifically:

#geom	Specify icon geometry
-bd n	Border width
-bg colour	Background colour
-display name	Name of display to open
-geometry geom	Specify window geometry
-help	Summarise options
-iconic	Open window as an icon
-name name	Specify name of window
-preopened display_id:window_id	Specify device and window id's
-synchronise	Synchronise with server (debugging only)
-title title	Specify title of window

Where `geom` is a standard geometry string of the form `WxH+-X+-Y`, and the `preopened` option is for a programme calling SM non-interactively. All options may be abbreviated, so

```
device x11 -i #-1+1 -g 512x512+100+100
```

specifies that the graphics window be created as an icon in the top right hand corner of the screen, and that the real window should be 512×512 and positioned near the top left corner. On hardware that doesn't support a backing store (or if you have chosen to disable a backing store when compiling the X11 driver) the screen will only be refreshed when it is active – the simplest way to get a refresh is to reopen it (`dev x11`).

On a Unix-PC `DEVICE upc` opens a window of 304 by 192 pixels, which is about 4 by 4 inches. To quote the author (Peter Teuben, teuben@astro.umd.edu),

> (1) This whole Unixpc version is an experimental version, take it as is, it works on my configuration, but may not work on yours.
>
> (2) I'm playing with allowing a second and third parameter to the upc device name, which would allow you to change the default size of 304 by 192 pixels (The size in X must be a multiple of 16 though). Right now I have it check environment variables `YAPP_X` and `YAPP_Y`, but this may not work satisfactorily.
>
> (3) I spawn windows using the public domain program 'wlogin', this may be of some importance if your 'upc' device in SM fails.

I don't have a Unix PC, so I can't work on this driver.

Syntax: DO `variable = start, end [ , incr ] { commands }`

While the value of `$variable` runs from `start` to `end`, the commands are executed. The optional increment defaults to 1. It is not possible to change the value of the loop variable inside a loop (or at least it has no effect on the next iteration). To break out of a loop you have to break out of the current macro as well with RETURN.

For example,

```
DO i=1,10,0.5 { WRITE STANDARD $i }
```

will write `1 1.5 2 2.5 ... 10` to the terminal. The commands may be spread over several lines.

Syntax: DOT

Draw a point at the current location (set by RELOCATE, DRAW, etc.) in the style determined by PTYPE. The point's size and rotation are governed by EXPAND and ANGLE.

To insert dots into labels, it may be easier to use the 'TEX' definition `\point` or `\apoint` which inserts a dot of a specified PTYPE into a string, see Appendix X for details.

Syntax: DRAW `#1 #2`
DRAW `( #1 #2 )`

Draw a line from the current position (set with, for example RELOCATE) to (#1, #2) in user coordinates. If the parentheses are present, use screen coordinates.

Syntax: EDIT `function key_strokes`

Bind a function to a set of `key_strokes` for the editor. For example, `EDIT refresh ^R` makes the ^R key refresh the screen. A complete list of functions is given in the 'Changing Key-Bindings' section in the main part of this manual (see under 'bindings' in the index). Each character in the key sequence can be specified as a character, *e.g.* '`a`' or the single character '`^A`', as '`^c`' representing the single character `^c` as the two character sequence '`^`' followed by '`c`', or by '`\nnn`' where `nnn` is an octal number (*e.g.* `EDIT refresh \022`).

In order to use multiple key sequences (*e.g.* `^A^B^C`) you must first undefine any sub-sequences, in this case `^A` and `^A^B`, by making them illegal – `EDIT illegal ^A`.

See `READ` for how to define a set of keys from a file, and `KEY` for how to define keys to execute commands.

Environment Variables

SM environment variables are defined in the file **.smongo**, which is first sought in the current directory, and then in your home directory. Alternatively, you can specify the name of the environment file using the '-f' flag on the command line, although this may fail if you try to use a raster device.

Each line of the file is taken to consist of a variable name, and the rest of the line which is taken to be its value. Any variable may be accessed using the DEFINE `name` : command, which defines `name` from the environment file.

Some entries in the environment file are special to SM, although you are free to use them to your own ends as well. These are:

background	The background colour for plots
case_fold_search	If non-zero, make searches case insensitive
default_font	The default font for labels
device	Your initial graphics device
edit	A file of keybindings to read
file_type	The current type of 2-D image for the IMAGE command
filecap	The filecap file to use (defaults to the same as graphcap)
fonts	The name of the binary fonts file
foreground	The foreground colour for plots
graphcap	The graphcap file to use
help	The help directory
history	The length of the history list (default 80)
history_char	The character used to recall commands (default ^)
macro	The default macro directory
macro2	Your private macro directory
overload	If non-zero, overload some commands
printer	The default printer (see hcopy and hmacro)
name	The name SM calls you by
temp_dir	The directory for temporary files
termcap	The file describing terminals to SM
TeX_strings	Use TEX-style strings for labels

Under Unix, you should omit the `termcap` entry, or point it at '/etc/termcap'. Also under Unix, SM knows how to look up your name, so you can omit the `name` entry. If you try to use a name with more than one word, SM will use the first so you'll have to call yourself 'my_lord' rather than 'my lord'.

Some of these are used directly by SM (*e.g.* `help`, `fonts`, but some are merely used by the `startup` macro to set the initial value of SM variables (*e.g.* `TeX_strings`, `file_type`). Other names may be used by the default `startup` macro, e.g. `macro2` to specify a private macro directory or `term` to specify the terminal that you are using. See the discussion of `startup` under 'useful macros'.

Syntax: ERASE

ERASE erases the graphics screen if plotting to a terminal, or reinitialises a hardcopy plot (sets the vector count to 0). The macro `era` erases the screen without appearing on the history buffer.

You may be able to erase individual lines with LTYPE ERASE, if you can you should look at the macro `undo`.

Syntax: ERRORBAR `WORD1 WORD2 expr INTEGER`

ERRORBAR is analogous to POINTS; it draws error bars on all points defined by vectors `WORD1` and `WORD2`, where the length of each errorbar is set by the corresponding value in `expr`. `INTEGER` is 1 to put the bar along the +x direction, 2 for +y, 3 for -x, and 4 for -y. Use EXPAND to govern the size of the caps. In fact, instead of either or both of `WORD1` and `WORD2` you can use an expression in parentheses, for example ERRORBAR `(lg(x)) (lg(y)) 120 1`.

See also the macros `ec` and `err` for backwards compatibility with Mongo, and `error_y` to draw a pair of errorbars in the y-direction. There is a macro `logerr` to draw errorbars on logarithmic plots.

Syntax: EXPAND `expr`

EXPAND expands all characters and points, its default is 1.0. Note that the EXPAND factor is used in determining the plot window size in the WINDOW command. This means you should declare your EXPAND size to SM (if other than the default) before you use WINDOW.

If EXPAND is set to exactly 1, and ANGLE is exactly 0, then SM will use hardware fonts, when available, in writing labels. This is faster, but if you don't like it say "EXPAND 1.00001", or "ANGLE 0.00001", or use a \r explicitly to select the roman font.

EXPAND can in fact be given a vector of values, which are used for each point in a POINTS command. This supercedes the use of a fractional PTYPE (although we still support it as a quaint anachronism). Using vectors for both ANGLE and EXPAND makes it easy to draw a vector field, see (for example) the `vfield` macro.

Syntax: FFT `n pexpr1 pexpr2 WORD1 WORD`

Fourier transform 2 vectors (treated as the real and imaginary parts of a complex vector), returning the answer in the two vectors `WORD1` and `WORD2`. The input vectors may be the names of vectors or expressions in parentheses. The direction is specified by `n`, either +1 for a forward transform, or -1 for an inverse.

The dimension of the vectors need not be a power of 2, but the transform is more efficient if it is. The worst case is when `n` is prime, in which case this command performs a slow Fourier transform in $O(n^2)$ time.

Syntax: FOREACH `variable ( list ) { commands }`

The value of `variable` is set to each element of list in turn, and then the commands are executed. An example would be

```
FOREACH var ( alpha 2 gamma ) { WRITE STANDARD $var }
```

which would write `alpha`, `2`, and then `gamma` to the terminal.

Syntax: FORMAT `[ x-format-string y-format-string ]`

Allow the user to specify the axis tick label formats. The format should be given as a standard C (e.g. %4.1g) or Fortran (e.g. F10.4). This format will be in effect until unset by issuing the FORMAT command with no argument, in which case SM will figure out the best format for you, or by issuing a new FORMAT command with new format strings. It is possible that certain Fortran formats will not work as desired, in particular it is not possible to specify that '.1' be written '0.1'.

If a format is specified as "0", the format string is left unchanged; if it is given as "1", the default value is reinstated. The command FORMAT 1 1 is thus equivalent to FORMAT.

Syntax: GRID `[ INTEGER1 [ INTEGER2 ] ]`

Grid draws a grid at either major (`INTEGER1 = 0`) or minor (`INTEGER1 = 1`) tickmarks within a box. The default is `INTEGER1 = 0`. You can use `INTEGER2` to specify only drawing an x- or y-axis grid: if `INTEGER2` is omitted or 0, draw both x and y; if it's 1 only draw x; if it's 2 only draw y (3 is equivalent to 0).

Syntax: HELP [word]

The HELP command tries to help you with word. If possible, it prints the entry from the help directory specified in your .sm file, the definition of word if it's a macro, the value of word if it's a variable, and the HELP string if it is a vector. If none of these are defined, HELP confesses defeat. You might want to use the abbreviation h which will not appear on your history list (or you could overload help itself).

If word is omitted it is assumed to be HELP.

See also APROPOS and LIST.

Syntax: HISTOGRAM WORD1 WORD2 [IF (expr)]

HISTOGRAM connects the points in vectors WORD1 and WORD2 as a histogram. If the IF clause is present, only use those points for which expr (see the section on vector arithmetic) is true (*i.e.* non-zero).

In fact, either or both of the WORDs may be replaced by 'parenthesised expressions', *i.e.* expressions in parentheses. For example,

```
HISTOGRAM x (2 + y)
```

to plot x against 2 + y. There is a macro barhist for drawing bar charts. See Arithmetic for how to convert vectors of data into histograms, and SHADE for how to shade them.

Syntax: HISTORY [-]

List the current commands stored in the buffer. For details on the history system, see the appropriate section in the first part of this manual. With the optional minus sign, the history list is printed backwards which is probably what you want if you are thinking of it as a set of commands to repeat. It's possible to overload list to be a synonym for HISTORY, see 'overloading' in the index.

Syntax: IDENTIFICATION str

IDENTIFICATION puts the current date and time followed by str outside the upper right hand corner of the plot region. (Actually, identification is a macro, which RELOCATEs to a point above the right-hand axis, and half way between the top axis and the top of the page, and then writes a string with a PUTLABEL 4.) Note that the variable $data_file is set to the name of the current data file, and $date always expands to the current date and time.

Syntax: IF (expr) { list }
IF (expr) { list } ELSE { list }

If the expr is true (non-zero), then the list of commands are executed, otherwise the ELSE clause is executed. For various complicated reasons, the ELSEless command must end with a newline (or as usual a \n). One common use for IF tests is when the expression tests if a variable has been defined, e.g.

```
IF($?file_name == 0) { DEFINE file_name ? }
```

within some macro.

There are also commands using IF to define vectors conditionally (see SET), and to plot parts of vectors (See CONNECT, HISTOGRAM, POINTS).

Syntax: IMAGE file
IMAGE file xmin xmax ymin ymax
IMAGE CURSOR
IMAGE DELETE

Read an image from file, optionally specifying the range of coordinates covered by the data values. If you do not specify them they will be taken to be 0 nx-1 0 ny-1 where nx and ny are the dimensions of the image.

IMAGE CURSOR is identical to the CURSOR command (which see), except that value of the image under the cursor is returned in addition to the position.

IMAGE DELETE will forget the current image and levels.

The file format is specified using a **filecap** file, similar to **graphcap**, and the entry to use in this file is given by the variable file_type (see Appendix V). The file is unformatted, and should start with two integers giving the dimensions of the data array, followed by the data values written row by row.

The current entries in **filecap** support files written from C, or from fortran in one of a variety of ways. For C programmers, DEFINE file_type C, the file should be written with open/write/close. For Fortran, there are a variety of options depending on operating systems and the details of how the file was opened. Under Unix, simply DEFINE file_type unix. Under VMS you have a choice. You can either specify the record length to be 512 (recl=512) and choose file_type vms_var, or set recordtype='fixed' choose recl to be the x-dimension of the array and define file_type to be vms_fixed. The one thing that you should not do is to omit the recl parameter entirely. By default, data is taken to be real (float in C), but this can be overridden in the filecap entry for a file type. There is also an entry for FITS files (FITS is the 'standard' image transport format for astronomical images). If you want to use a different file type, read Appendix V or see your local SM Guru.

So under VMS either your code should look like (file_type = **vms_var**)

```
      integer i,j
      real arr(100,10)
c
      open(2,file=filename,form='unformatted',recl=512)
      i = 100
      j = 10
c now write your data into arr
      write(2) i,j,arr
      end
```

or, with file_type = **vms_var**

```
      integer i,j
      real arr(100,10)
c
      open(2,file=filename,form='unformatted',recl=100,recordtype='fixed')
      i = 100
      j = 10
c write your data into arr here
      write(2) i,j
      do 1 j=1,10
          write(2) arr(1,j)
1     continue
      end
```

Under Unix, either of these programme fragments would work after omitting the record information from the open statement.

See also SET for how to extract a cross-section into a vector, and DEFINE for defining a variable from the image header.

Syntax: KEY

KEY **key string**

Define a key to generate a string. This is most often used simply to save typing some common command such as **edit_hist**. With the command KEY, you are prompted for the **key** to define, and the string. Because you are not using the history editor when you type the key, you can simply hit the key that you want defined, type a space, and then type the string terminated by a carriage return. The other form, where the whole command appears on one line, is probably more suitable for use in a macro such as your private startup macro (see under **startup2** in the index). If you try entering it at the keyboard any special characters in **key**, such as ESC, will be interpreted by the history editor so you'll probably have to quote the **key** with ^Q or ESC-q. Alternatively you can use ^ and printing characters , or octal numbers, to represent the escape characters in the same way as for EDIT (*q.v.*). If **key** is given as **pf#** or **PF#** (where **#** is 1, 2, 3, or 4) it will be interpreted

as a special function key on your keyboard in a terminal-independent way (see the description of termcap in Appendix VI to see how these keys are defined). KEY definitions are listed along with other key bindings by the LIST EDIT command.

If the `string` ends in a \N, it will be executed the moment that the key is struck. (Note that this is \N not \n, which would have been interpreted as a newline.)

Only 10 keys can be defined, after that you'll start overwriting earlier definitions.

Syntax: LABEL `str`

LABEL writes the string `str`, which starts one space after LABEL and continues to the last non-space character, at the current location (set by RELOCATE, etc). You can of course use quotes to include trailing white space. The string's size and angle are determined by EXPAND and ANGLE.

As for all labels, you have a choice of two text formats, the traditional one, and one based on TEX. If you want the latter, define the variable `TeX_strings`.

In 'traditional' labels, the character "\" is an escape character and causes the following actions:

\\x	set mode x	\x	set mode x for next char only
\r	roman font	\g	greek font
\s	script font	\t	tiny font
\i	toggle italics	\u	superscript
\d	subscript	\e	end string
\#	expand	\\\	write a \

'#' is any digit, optionally preceded by a '-'. These change the size of the succeeding characters by multiplying the current value of EXPAND by a factor of $1.2^{\#}$, *e.g.* `\2` increases by 1.44, `\-3` decreases by 0.58. These take place in addition to the expansion produced by going up or down (`\u` and `\d`), or setting EXPAND. As special cases, `\s` followed by a space is a narrow space, while `\g` followed by a space is a negative space (*i.e.* backspace the width of a space).

In TEX mode the same fonts are available (plus a 'bold' font), and in addition you can use escapes such as `\beta` to write a β, or `\point` to insert a point of a given PTYPE. Sub- and super- scripts are specified as _ and ^, just like real TEX. Grouping is achieved by using braces, *e.g.* `\int_{-\infty}`. For a complete description, and list of control sequences, see Appendix X.

If you want to change the font used for labels, define the variable `default_font`, either interactively (`DEFINE default_font oe`), or by putting a line in your **.sm file**:

`default_font oe`. This affects axis as well as regular labels and only works if you use `TeX_strings`.

If EXPAND is set to exactly 1, and ANGLE is exactly 0, then SM will use hardware fonts, when available, in writing labels. This is faster, but if you don't like it say "ANGLE 360", or use a \0 to select a font with (explicitly) no expansion. To affect the axis tick labels too, using the AXIS or BOX commands, you'll have to say "EXPAND 1.0001" or somesuch. Rather than always expanding your plots, you could ask your SM Guru to edit the **graphcap** file to prevent a given device (usually a printer) from ever using hardware fonts. Tell her to look in Appendix II. If she won't oblige, you can define your own device in your own graphcap file, and put yours first in the **.sm** file. For example, my **.sm** file includes the line

```
graphcap    /d/rhl/graphcap /d/SM/graphcap
```

and the file **/d/rhl/graphcap** looks like:

```
#
# Private overrides for RHL:
#
postscript_rhl|postscript + no hardware fonts:\
    :TB@:TE@:tc=postscript:
```

Then I set `$printer` to `postscript_rhl` (also in **.sm**) and all is well.

An alternative is to specify the device as `postscript :TB@:TE@:tc=postscript:` which is perhaps simpler (you'd just define your value of `printer` properly).

Syntax: LEVELS `WORD`
LEVELS `expr`

Set the levels for the CONTOUR command to be the values of the vector `WORD` or to the values of the expression.

Syntax: LIMITS `WORD WORD`
LIMITS `WORD Y1 Y2`
LIMITS `X1 X2 WORD`
LIMITS `X1 X2 Y1 Y2`

LIMITS sets the coordinates of the plot region. All coordinates in RELOCATE, DRAW, etc, are referred to these limits. The various forms specify explicit limits for the x or y axis (`X1 X2` or `Y1 Y2`), or default (specify the name of the vector to be used).

In fact, either or both of the `WORD`s may be replaced by 'parenthesised expressions', *i.e.* expressions in parentheses. For example,

```
LIMITS 0 5 (ln(y))
```

will scale the y axis according to the logarithm of vector y (but *not* produce a logarithmic axis - see TICKSIZE for this capability).

If the two limits specified for an axis are the same, the limits for that axis will not be changed.

You can specify that the limits on one or both axes have a desired range using the RANGE command. This command affects the performance of the LIMITS command. If a non-zero RANGE has been set, LIMITS will ensure that the upper and lower limits differ by that amount. (e.g. after RANGE `2 0` , LIMITS `0 1 0 1` is equivalent to LIMITS `-0.5 1.5 0 1`). If you specify a vector, the range is centred on the median value. If you have specified a range, and then ask for logarithmic axes with TICKSIZE, you may get complaints that logarithmic axes are impossible. Simply unset RANGE, and the problem should go away.

Syntax: LINES `INTEGER INTEGER`

Use only lines `INTEGER1` to `INTEGER2` from the current data file (specified with the DATA command). If VERBOSE is greater than 0, the lines actually read will be reported. LINES 0 0 will use the entire file, which is also the default following a DATA command. The variables `$_11` and `$_12` will be set to the first and

---

Syntax: LIST DEFINE `[ begin end ]`
LIST DEVICE
LIST EDIT
LIST MACRO `[ begin end ]`
LIST SET

list all the currently defined variables (DEFINE) or macros (MACRO), optionally only within the range `begin - end` . If VERBOSE is 0 macros beginning ## won't be listed.

LIST EDIT will list all the keybindings. If VERBOSE is 0 only the keys that don't generate themselves are listed (*i.e.* because A is bound to A it isn't listed). If VERBOSE is 1, in addition all non-printing keys are listed, and if VERBOSE is 2 or greater all keys are listed. Both the EDIT and the KEY bindings are listed.

LIST DEVICE will list all the devices known to SM. The possible stdgraph devices (default devices defined in graphcap) are listed with each device on an (indented) line, first the primary name, then a list of aliases in parentheses, then a full name.

LIST SET lists all currently defined vectors. For each vector the name, the dimension and the `HELP` field are given. See `SET` for how to set the latter.

For a list of the history buffer use HISTORY (macro `lis`), to list a macro use HELP (macro `h`). It can be useful to overload 'list' so that it doesn't appear on the history list, and so that 'list' by itself corresponds to the command HISTORY (this is done for you if you use `set_overload` or put `overload` in your **.sm** file).

Syntax: LOCATION `GX1 GX2 GY1 GY2`

Set the physical location of the plot. The plot region is the rectangle inside the box drawn by BOX. Vectors and points are truncated at the bounds of the plot region. LOCATION specifies (in device coordinates) where the plot region is located. LOCATION can be used to make an arbitrary size and shape plot, providing that you want it rectangular.

Because all devices have the same coordinate system in SM (0-32767), this command is considerably more useful than it used to be. The default LOCATION is 3500 31000 3500 31000.

While you are using WINDOW (*q.v.*), LOCATION commands have no effect. SM remembers them, however, and obeys the most recent one when you are finished with WINDOW.

See the RELOCATE `( x y )` command to draw labels outside the plot region, and DRAW `( x y )` to draw lines there.

Logical Operators

The following logical operators are allowed on vectors and scalars in SM, where non-zero means true:

expr == expr	Equal to	expr ! = expr	Not equal
expr < expr	Less than	expr <= expr	Less than or equal
expr > expr	Greater than	expr >= expr	Greater than or equal
expr && expr	Logical and	expr || expr	Logical or

Only == and ! = are allowed for string valued vectors. All arithmetic vectors test unequal to all string-valued vectors.

As in C, == has a higher precedence than &&, which in turn has higher precedence than ||.

Note that there is also a ternary operator, `(expr1) ? expr2 : expr3` which has the value `expr2` if `expr1` is true, and `expr3` if it is false.

See 'arithmetic' for the arithmetical operators.

Syntax: LTYPE **INTEGER**
LTYPE **ERASE**

All lines except for axes, points, and characters are drawn with line type **INTEGER**, meaning:

0	solid	1	dot
2	short dash	3	long dash
4	dot - short dash	5	dot - long dash
6	short dash - long dash		

the default is a solid line, LTYPE 0.

LTYPE ERASE and will erase any lines that are redrawn (*e.g.* LTYPE 0 BOX LTYPE ERASE BOX will first draw a box, and then erase it). Not all devices can support erasing individual lines, if yours doesn't you'll have to ERASE the whole screen. A convenient way to use LTYPE ERASE is the **undo** macro. (in fact, LTYPEs 10 and 11 are used to implement LTYPE ERASE, LTYPE 10 to start erasing, LTYPE 11 to notify a device that you've finished doing so).

Syntax: LWEIGHT **INTEGER**

Set all lines to have a weight of **INTEGER**, where the bigger the blacker.

Syntax: MACRO **EDIT name**
MACRO **LIST [begin end]**
MACRO **name [narg] { macro }**
MACRO **name DELETE**
MACRO **name #1 #2**
MACRO **READ file**
MACRO **WRITE file**
MACRO **DELETE file**
MACRO **WRITE name [+] file**

MACRO EDIT name allows you to edit a macro. All the commands available to the history editor area available (including the ^ history), except that ^M inserts a line before the cursor, ^N and ^P get the next and previous lines respectively, and ^V and ESC-v move forwards and backwards 5 lines at a time. To exit use ^X (or whatever you have bound to **exit_editor**). The macro need not exist, and both its name and number of arguments can be changed by editing the zeroth line of the macro (^P from the first line. If this line is corrupted, or deleted, no changes are made to the macro when you exit. If the number of arguments is negative, the macro will be deleted when you exit.) You may prefer to use the macro **ed** instead

of MACRO EDIT, as it doesn't appear on the history list and, if invoked without a macro name will edit the macro that you edited last. Incidently, `hm` ('help macro') will list the last macro that you edited with `ed`. The keybindings may be changed with READ EDIT.

LIST MACRO lists all currently defined macros, or all those which are between `begin` and `end` alphabetically (asciily). If VERBOSE is 0, macros starting with ## are not listed.

MACRO `name [narg] { macro }` defines `name` to be `macro`, where `name` is a single word, and `macro` may be anything. A macro is invoked by typing its name. The optional `nargs` is the number of arguments the macro expects, default 0. If the macro's body is defined to be `delete`, the macro is deleted. MACRO `name` DELETE also deletes a macro. Arguments are referred to as $1, $2, ... $n, up to a maximum of $9. $0 gives the name of the macro. If the number of arguments is declared as more than 9, the macro is taken to have a variable number of arguments, up to the number declared modulo 10. When called, all the arguments must appear on the same line as the macro itself. This line may, as usual, be ended with an explicit \n. The macro can determine whether it has been supplied a given argument by using the `$?` construction (see DEFINE). It is also possible to change the values of arguments using DEFINE just as usual, and even to DEFINE arguments that you didn't declare. These are temporary variables, local to the macro, and will disappear when you exit the macro.

MACRO `name #1 #2` defines macro name to consist of lines #1 — #2 of the history buffer.

MACRO READ `file` reads the macros in `file` and defines them. See RESTORE for how to also restore the history buffer from macro `all`.

MACRO DELETE `file` has the effect of deleting all macros defined in `file`.

MACRO WRITE `file` writes all currently defined macros to `file` in alphabetical order.

MACRO WRITE `name [ + ] file` writes the macro `name` to `file`. If the `+` is specified, or the file is the same as for the previous use of this command, the macro is written to the bottom of the file, otherwise the file is created.

Syntax: MINMAX `min max`

Set variables `min` and `max` to the the maximum and minimum values of an image read by the IMAGE command. Only that portion of the image within the current LIMITS is examined. This may be useful for setting contour levels, or doing a halftone plot (which is not currently implemented).

For example, the commands:

```
MINMAX min max
SET levs = $min,$max,($max-$min)/9
LEVELS levs
```

will choose a set of 10 levels which cover the complete range of the data.

Syntax: NOTATION XLO XHI YLO YHI

Set axis label format (exponential or floating). By default, all numbers between 1e-4 and 1e4 are written as floating point numbers, and all numbers outside this range are written with an exponent. This corresponds to a **NOTATION -4 4 -4 4** command.

If you set XLO=XHI and/or YLO=YHI, all values on that axis will be plotted using exponents.

Syntax: OVERLOAD **keyword INTEGER**

Allow "keyword" (in lowercase) to be used as a macro name if integer is non-zero. For example,

```
overload set 1 overload define 1
macro set { DEFINE } macro define { SET }
```

would interchange the meanings of the SET and DEFINE commands. The uppercase forms of the keywords retain their usual meanings. **overload set 0** would reinstate the usual meaning of set. You may be surprised by the effects of overloading certain keywords. For example, if you overload "help" to mean "DELETE HISTORY HELP", then "set help vec help_string" won't work (you'd have to say "set HELP vec ...").

This command is intended to be used for changing the default action of commands, rather than for a wholesale renaming of keywords! A more practical example than the above would be

```
overload erase 1 macro erase { del1 ERASE }
```

to prevent erase commands from appearing on the history list. See the macro **set_overload** for a set of definitions like this. It can be automatically executed by including an "overload" line on your **.sm** file.

Syntax: POINTS WORD1 WORD2 [IF (expr)]

POINTS makes points of the current style (PTYPE), size (EXPAND), and rotation (ANGLE) at the points in vectors **WORD1** and **WORD2**. If the IF clause is present, only use those points for which **expr** (see the section on vector arithmetic) is non-zero. In fact, either or both of the **WORD**s may be replaced by 'parenthesised

expressions', *i.e.* expressions in parentheses. For example,

```
POINTS x (lg(y))
```

to plot x against the logarithm of y.

In case you ever need to know, the distance from the centre of a point to a corner is 128 screen units when unexpanded, if the ASPECT (*q.v.*) ratio

---

Syntax: PRINT `[ file ] [ 'format' ] { list }`
PRINT `[ file ] [ 'format' ] < list >`

Print the vectors specified by `list` to `file`, if `file` is absent, print to the terminal (the output is paged, sort of). The name of each vector is printed at the head of the appropriate column. If the output is going to a file, each line of the header starts with a '#', so the file can be read without using the LINES command.

The optional format string is of the type accepted by the C function 'printf', and you should see a book on C (or maybe the online system manual or help command) for more details. Basically, the format string is copied to the file with format specifiers beginning with `%` signs replaced by the numbers that you want printed. The format specifiers to use are the floating point ones, `%e` (exponential), `%f` (floating point), and `%g` (computer's choice, good for integers) or `%s` for strings. Fields are right justified by default, you can insert a - just after the `%` to left justify them. A `%` may be written as `%%`, and a tab as `\t`. Lines are *not* terminated by a newline by default, you have to write them explicitly as `\n`.

For example,

```
SET x=1,10 SET y=x**2
PRINT file '%10f (%10.2e)\n' { x y }
```

will produce

```
#........x............y
#
..1.000000.(..1.00e+00)
..2.000000.(..4.00e+00)
..3.000000.(..9.00e+00)
(etc.)
```

where I have replaced each space by a . for clarity. If you say

```
PRINT '%g ' { x }
```

you will get

```
..........x
1.2.3.4.5.6.7.8.9.10.
```

Syntax: PROMPT `new_prompt`

The current prompt is replaced by `new_prompt`. Any occurrences of the character

Syntax: READ WORD INTEGER
READ { WORD INTEGER WORD INTEGER ...}
READ ROW WORD INTEGER
READ EDIT WORD
READ OLD WORD WORD

READ WORD INTEGER reads a column of data from the file specified by the DATA command, using the lines specified by LINES. Columns may be separated by white space (blanks or tabs) or by a comma, or by some combination of the two. It's OK if some of the columns contain text, providing that you don't try to read them. You can read text columns into string vectors, as described in the next paragraph. The data is read into the vector WORD, which will be created, from column INTEGER. Any field beginning with a * is taken to be 'empty', and is assigned the value 1.001e36. Any line beginning with a # is skipped over (and printed if VERBOSE is greater than 1), any line ! is skipped and always written to the terminal. Long (logical) lines may be spread over several (physical) lines by ending the line with a '\'; no line may exceed a total of 1500 characters.

You can optionally specify a type of vector by adding a suffix onto the integer; '.f' (the default) means floating point, '.s' means string-valued. String valued vectors can be used as input to PTYPE commands, or simply for reading columns from data files that you want to PRINT.

READ { WORD INTEGER WORD INTEGER ... } is the same as repeating READ WORD INTEGER for each vector, but more efficient as it only has to read the file once.

```
READ { x 1 y 5.s z 2.f }
```

will read columns 1 and 2 into floating point vectors x and z, and column 5 into string-valued vector y.

If INTEGER is invalid (<= 0, or > 40), the contents of the file are written to the standard output. READ ROW is very similar, but the values are read from row INTEGER of the file. The same type qualifiers are allowed as for reading columns.

READ EDIT WORD reads a new set of keybindings from the file WORD. The format and syntax are given under History in the introduction.

READ OLD WORD1 WORD2 defines macro WORD1 to be the the contents of file WORD2. This is provided for compatibility with Mongo, see Appendix IX and the read_old macro. You no longer need use read_old to read SM history files, use RESTORE instead.

If VERBOSE (*q.v.*) is greater than 0, the lines actually read will be reported.

Syntax: RELOCATE `X Y`
RELOCATE `( X Y )`

The first form sets the current position to (`X,Y`) in user coordinates without drawing a line. The second (with parentheses) sets the position in 'screen' coordinates, *i.e.* 0-32767. The current position is used by the DRAW, LABEL, and PUTLABEL commands.

Syntax: RESTORE `[` filename `]`

Restore all the current macros, variables, and vectors from file `filename` (if omitted the default is to use the value of `save_file` in your **.sm** file, or failing that **sm.dmp**). In addition, the current history buffer is replaced by the macro `all` if defined in the RESTOREd file.

The file should have been written by the SAVE command, and RESTORE will treat any other file type as if it were a SM history file and add its commands to the end of the current history list.

If VERBOSE (*q.v.*) is greater than 0, some extra information is printed.

Syntax: RETURN

Return from the current macro, which includes breaking out of DO and FOREACH loops. If you are not executing a macro, simply return to the prompt (this is more or less equivalent to typing ^C).

A RETURN can be useful while playing with fiddling with data interactively. For example, if you want to playback a set of commands, but then do other things when the plot has appeared, you could put a RETURN after the desired part of the playback buffer. (This doesn't work quite the way that you might naïvely think. `Playback` works by defining a macro `all` from the history list, and then executing it. The RETURN is actually returning from this macro, rather than directly from the command list, but the effect is the same. If RETURN always returned directly to the prompt, macros such as `hcopy` wouldn't work.)

If VERBOSE is 2 or more, the name of the macro being returned from is output.

If the very last command in a macro is RETURN then the RETURN will take place, not from the desired macro, but from where the macro was called from. You can work around this by putting a space after the RETURN, or simply omitting it as it isn't doing anything anyway. If a RETURN comes last on a history list, this problem will lead to macros such as `hcopy` *not working correctly.*

Syntax: SAVE [filename]

Save some or all of the current macros, variables, and vectors in file `filename` (if omitted the default is to use the value of `save_file` in your **.sm** file, or failing that **sm.dmp**). The current history buffer may also be saved, as the macro `all`. You are prompted for whether you want to save variables, vectors, and macros (which includes `all`, the current playback buffer). Macros beginning ## are *not* saved, as they are assumed to be system macros. Variables and vectors whose names start with a '_' are assumed to be temporaries, and are not saved either.

The opposite to SAVE is RESTORE *q.v.*. You may want to use the MACRO DELETE WORD command to undefine macros from *e.g.* the **utils** macro file. See, for example, the macro `sav` (which can be overloaded).

If VERBOSE (*q.v.*) is greater than 0, some extra information is printed.

Syntax: SET `name = expr`
SET `name = { expr }`
SET `name = expr` IF `( expr )`
SET `name = expr1, expr2 [, expr3 ]`
SET `name = expr1 ? expr2 : expr3`
SET DIMEN `( name ) = INTEGER`
SET `name = WORD ( [ WORD [ , WORD ...] ] )`
SET `name [ s_expr ] = s_expr`
SET HELP `WORD [ rest ]`

Conduct various operations on vectors of data. The simplest, SET `name = expr` sets vector `name` to be equal to the expression `expr`. If the IF clause is present, `name` will only contain those elements of `expr` for which it is true (non-zero). A special case of an expression is simply a list of values within braces. For string-valued vectors, the only allowable expressions are a string-valued vector, the CONCATenation of two string vectors, or a string in single quotes (*e.g.* SET `s='Hello, World'`).

With expressions separated by commas the SET command is like a DO loop, setting `name` to be the values between `expr1` and `expr2`, at increments of `expr3` (which defaults to 1).

The command with ? and : is similar to the C ternary operator. If `expr1` is true, take the corresponding value of `name` for `expr2`, otherwise use `expr3`. This command is worth learning, as it can often be used to replace a DO loop.

If you have a DO loop that calculates each element of a vector in turn, something that is possible if inefficient in SM, you need to define a vector before you use

it. This can be done with the SET DIMEN(`name`) = `INTEGER`, which also initialises it to 0. Thus DEFINE `i` (DIMEN(`x`)) SET DIMEN(`y`) = `$i` is equivalent to SET `y = 0*x`. (*Note that this is a change*; you used to be allowed to use expressions as the dimension). You can optionally specify a qualifier to the dimension, in just the same way that you can specify a qualifer to a column in a READ command, so SET DIMEN(`s`) = `10.s` declares a 10-element string-valued vector.

SET `name = WORD ( [ arg [ , arg ... ] ] )` allows you to use a macro as a sort of function definition. Within the macro WORD any assignment to `$0` has the effect of assigning to `name`, and the other arguments behave as normal. The arguments `arg` can be words or numbers (but not general expressions) and are separated by commas. *Note that this is a change to the syntax of this command!* Previously only one argument was permitted, but it could be an expression, and the result was returned by assigning to `$1` in a rather confusing way.

SET `word[s_expr] = s_expr` sets element `s_expr` ('scalar expression') of vector `word` to be a scalar expression. The first `s_expr` is converted to an integer before being used as an index. Note that arrays are subscripted with [ ] not (), and that, as always, indices start at 0 not at 1.

SET HELP sets the help string for a vector; the rest of the line is read, and will be returned in response to a HELP `WORD` request.

Let's look at some examples.

```
SET y = $v1 + 5.0 * x
```

This sets each element of the vector `y` to be the value of the scalar `$v1` plus 5.0 times the corresponding element of the vector `x` (assuming that `x` has been defined previously)

```
SET data_set_1 = lg(x) IF ( lg(x) > 0)
```

This sets the elements of the vector `data_set_1` to be the (common) logarithm of the corresponding element of the vector `x`, if that logarithm is > 0. Thus `data_set_1` will in general be of smaller size than `x`.

```
SET data = (lg(x) > 0) ?  lg(x) :  0
```

In this case, `data_set_1` will be the same size as `x`, and any elements of `data_set_1` where the corresponding element of `x` is less than or equal to 1, will be set to 0.

```
SET vec = 4*{ 1 1.5 2 2.5 3 }
```

will define a vector `vec`, with 5 elements, with the values given by four times those in the list.

```
SET vec = 1,12,2
```

an alternative way of defining the same values.

```
SET i = { 2 3 }
SET x = vec[i]
```

will set the vector `x` to have be `8 10` (*i.e.* `vec[2]` and `vec[3]`).

```
MACRO pow 2 { SET $0 = $1 ** $2 }
SET vec = pow(vec , 3)
```

cube the vector `vec`.

```
SET vec[0] = 2*pi
```

Change your mind about the first element of `vec`.

```
SET HELP pam Wichita, Kansas, July 7, 1953
```
will set the help string for vector pam to be the string Wichita, Kansas, July 7, 1953, so when you type HELP pam, this string will be printed out.
```
SET rhl=Robert
```
defines a string vector with one element.
```
SET DIMEN(rhl) = 10.s
```
defines a string vector with ten elements (all blank), while
```
SET rhl={Robert Horace Lupton}
```
defines a string vector with initialised elements, and
```
SET rhl[1]=Hugh
```
corrects it.

If you have an image defined (using the IMAGE command), you can extract a cross-section using the SET name = IMAGE (expr , expr) command. The two expressions give the (x,y) coordinates where you want the image to be sampled. For example,
```
SET x=0,1,.01 SET z=IMAGE(x,0.5)
```
will extract a horizontal cross section through an image.

See the CURSOR command for defining a pair of vectors using the cursor to mark the points, and SPLINE for how to fit splines to pairs of vectors.

Syntax: SHADE INTEGER pexpr pexpr
SHADE HISTOGRAM INTEGER pexpr pexpr

('Pexpr' is the name of a vector, or an expression in parentheses, *e.g.* SHADE 1000 x (sqrt(y))).

Shade 'inside' a curve defined by the expressions. The shading is rotated through the current value of ANGLE, and lines are spaced INTEGER apart (screen coordinates, so the full screen is 32768 across). If INTEGER is 0, the lines will be drawn as close together as the device allows, simulating an area fill. This is a very inefficient way to fill areas, made only slightly better by specifying a large LWEIGHT on devices that support such things in hardware (you'll also get slightly jagged edges).

The meaning of 'inside' is that as the shading is done, from left to right taking the value of ANGLE into account, lines are drawn from every odd to every even crossing of the curve. The curve is considered as being closed by joining the first to the last point. If a shading line just touches the curve the algorithm may be confused, change INTEGER slightly, or adding 180 to ANGLE. Sometimes joining the ends of the curve may not be what you want, try using CONCAT to add points on the end yourself. For example,
```
SET x=0,10 SET y=x**2 LIMITS x y SHADE 1000 x y
```
looks like a new moon, but
```
SHADE 1000 (x CONCAT 10) (y concat -1e10)
```

shades beneath the curve, for ANGLE 0 that is. You could also try the macros scribble and shading in demos, *e.g.* type load demos scribble.

SHADE HISTOGRAM is similar, but it shades the histogram that would be drawn by HISTOGRAM from the same set of points. In this case the area to be shaded lies between the histogram and the line y=0. If this offends you, offset the whole graph and lie about the axes.

Syntax: SHOW

List the values of some of the internal variables, including current location and plot region limits in user and device coordinates, the value of the expansion and angle variables, the line type and weight, and the physical limits. Show is actually a macro, so you could modify it to your own ends, for example listing the current data file too.

Syntax: SORT { vector_list }

Sort the first vector in the list into increasing numerical order, and rearrange the others in the same way. The maximum number of vectors that can be sorted is 10. For example, following the commands

```
SET e = { 2 7 1 8 2 8 1 8 2 } SET p = { 3 1 4 1 5 9 2 6 5 }
SORT { e p }
```

the vectors e and p would be 1 1 2 2 2 7 8 8 8 and 4 2 3 5 5 1 1 9 6 . The order within the p vector is not defined when the e values are identical.

Syntax: SPLINE x1 y1 x2 y2

Fit a natural cubic spline through the points specified by vectors x1 and y1. The dimensions of x1 and y1 must be the same and must exceed 2, x1 must be monotonic increasing (use SORT if necessary). When the spline has been fit, take the points specified in vector x2, and fill the (new) vector y2 with the corresponding values. Linear interpolation is used beyond the ends of x1.

Syntax: TERMTYPE word [INTEGER]

Set the terminal type to be WORD. This has nothing to do with graphics, but is to do with the history and macro editors. WORD is case-sensitive. The properties of the terminal will be read from the termcap file, for details see Appendix VI. For most purposes you don't even need to use this command, as when SM starts up it reads the value of the environment variable TERM (under Unix) or logical variable (under VMS) it effectively issues a TERMTYPE command with its value as argument. If

you have a `term` entry in your **.sm** file this takes precedence over any `TERM` variable. For example, a `term` entry of `selanar -21` is equivalent to the command `TERMTYPE selanar -21`.

You also should not have to use the optional `INTEGER` argument, which specifies the number of lines that will appear at a time when LISTing things, as this information is usually derived from termcap. If you are using a window system, then termcap may be wrong and this argument may be useful. Another exception occurs when you wish to disable cursor motion to avoid having your graphs scrolling off the screen. If this concerns you, read Appendix VI.

Syntax: TICKSIZE `SMALLX BIGX SMALLY BIGY`

Determine tick intervals for BOX. `SMALLX` refers to the interval between small tick marks on the x axis, `BIGX` refers to the interval between large ticks and so forth. If `BIG` is 0, the axis routine will supply its own intervals according to the label limits. If `SMALL` < 0, the axis will have logarithmic tick spacing and BOX assumes that the limits are logarithms, e.g. -2 and 2 refers to limits of 0.01 and 100.

Negative values of `SMALL` and `BIG` are interpreted as specifying the tickspacing in the decade 1:10, and are scaled to fit the decades actually plotted. For instance, if you say

```
LIMITS 0 1 3 4
TICKSIZE -1 10 -0.1 1 BOX
```

then the x-axis will have small ticks at 2, 3, ..., 9 and big ticks at 1 and 10, while the y axis will have ticks at 1010, 1020, 1030, ... and big ticks at 1000, 1100, 1200, ... The most usual TICKSIZE is probably `-1 10`, and this may be written `-1 0` for backwards compatibility.

Occasionally you may want to use the same tickspacing in all decades of your plot. To do this make `BIG` negative also in which case the spacing used for the first decade plotted will be used for all decades. (Note that this means that if the axis is plotted backwards then the value from the largest decade will be used):

```
LIMITS 1.9 2.1 2.1 1.9
TICKSIZE -0.1 -1 -0.1 -1 BOX
```

this is a good way to make an axis very crowded!

Syntax: USER `integer string`

Call a function called 'userfn', passing the `integer` and the string as arguments, both are passed by address as if SM were written in fortran (string is passed as a NUL terminated C string, though).

This function is provided to allow users without C compilers to make additions to the main grammar, but whether it is really useful is a different matter. Currently,

if **integer** is 50258 you'll get a segmentation violation (on purpose). If you really want some new functionality, send us mail.

Variables

See DEFINE for details on SM's variables.

Syntax: VERBOSE **INTEGER**

Make SM produce output on what it is doing if **INTEGER** is > 0. Setting VERBOSE to 0 is a way of only listing 'important' (non-system) macros, and generally getting a little peace and quiet. It has the considerable disadvantage that you can think that you are reading data from files, while actually something is wrong. For this reason the default value is 1. A value of 2 or more is basically useful for debugging. If you get some nondescript syntax error and don't know where it is coming from, VERBOSE of 3 or 4 will trace your programme, and should help find the problem. The original error message will tell you which macro SM thinks it is processing when the error occurred but it will be wrong if the macro had been fully scanned when the error is detected. In this case it will report a parent of the current macro. The reason for this behaviour is related to why **RETURN** can return from the wrong place, and is discussed in Appendix I.

If you want to know the current value of VERBOSE you can use the SHOW command (actually a macro), or try

```
DEFINE verbose | echo Verbose:  $verbose
```

which is (of course) what SHOW does anyway.

If verbose is one or more SM will:
- Classify vectors as number- or string-valued
- Complain about things like division by zero, and logs of negative numbers the first time that they occur in a given expression
- Include all macros in macro listings, even those starting ##
- Indicate line where arithmetical errors occur
- List all non-printing key-bindings with LIST EDIT
- Note more than 40 curve crossings in SHADE
- Notify user when a ^C stopped the production of a hardcopy
- Output a little extra about RESTORE and SAVE
- Print lines beginning # while reading macro files
- Report which lines are read from a file using READ
- Tell a bit about format and size of IMAGE files
- Warn about zero-length vectors
- Whinge about missing fields with DEFINE var READ # #

if **INTEGER** is two or greater, then also :
- Announce when the do, foreach, input, or macro stacks are extended

Complain if an environment or SM variable is not defined
Echo lines in data files that start with #
List all key-bindings with LIST EDIT
Note attempts to unput off bottoms of buffers
Prompt for variables, even if they are on the macro buffer
Protest if you reference an undefined history number
Remind you that only a finite number of KEY commands can be processed
Repeat complaint about things like division by zero, and logs of
negative numbers every time that they occur
Report vectors of different lengths in PRINT
Say when a vector is used as a scalar.
Tell you when vectors are redefined.

if INTEGER is three or greater, then also :
List each macro name just prior to expansion.

if INTEGER is four or greater, then also :
Print each token as it is recognised along with its text. The number in
column 1 is the value of 'noexpand', the level of nesting of { } loops.
Show expansions of variables.

if INTEGER is five or greater, then also :
Print the contents of DO, FOREACH, IF, and MACRO command lists (and
also prompts for DEFINEing variables, and the values of multi-word variables).

If you set a negative verbosity, then if the parser was compiled with DEBUG defined, you'll get a veritable torrent of debugging information. Use another negative VERBOSE command to turn it off again.

Syntax: VERSION

Return a string identifying the version of SM in use. If you have any reason to communicate with SM's authors, we'll want to know which version you are running.

Whatis

WHATIS(something) has a value depending on what something is:

a number:	0	
not a number:	1	
	a macro:	set 02 bit
	a variable:	set 04 bit
	a vector:	set 010 bit
	a keyword:	set 020 bit

So if "aa" is the name of a vector, WHATIS(aa) has the binary value 011, or 9, whereas WHATIS(HELP) has the value 021, or 17, and WHATIS(1) is 0. There is a macro in **utils** called `is_set` that tests if WHATIS sets a particular bit, for example

```
is_set test kkk 1 if($test) { echo kkk is a macro }
```

tests if bit 1 (macro) is set for "kkk" and prints its findings.

Syntax: WINDOW `nx ny x y`

WINDOW makes the current plot location the window at (x,y), where there are nx windows across and ny windows up and down. WINDOW 1 1 1 1 resets the plot location to the entire plot area. The size and placement of the windows is decided by the value of EXPAND when the WINDOW commands are issued, so be sure that EXPAND has the same value for every window in a set. (It's used to figure out the axis labels, and spacings between boxes). While plotting to a given window you can of course change EXPAND to your heart's content.

If the number of windows in either the x or y direction is negative no space is left between the boxes in that direction (try DO i=1,3 { WINDOW `1 -3 1 $i` BOX }). It's possible to overload 'window' and 'box' to only label external axes in blocks of touching boxes.

The position of the windows pays no attention to the LOCATION used when the windows are first set up, and LOCATION commands are ignored until you are finished with your windows. When you finally say WINDOW `1 1 1 1` the most recent LOCATION is restored.

Syntax: WRITE STANDARD `string`
WRITE `WORD string`
WRITE HISTORY `WORD`

WRITE STANDARD writes a string, followed by a newline, to the standard output. The string is taken to be the rest of the line up to a carriage return (which may be written explicitly as \n). The macro `echo` is usually used as an abbreviation for this command. WRITE `WORD` is similar, except that the string is written to file `WORD`. If the filename is the same as the previous WRITE, the string is appended, otherwise the file is overwritten.

WRITE HISTORY `WORD`, writes macro `WORD` onto the end of the history list.

For MACRO WRITE, see under macros.

Syntax: XLABEL `str`

Write the label `str` centered under the x axis made by BOX. The string is taken to be the rest of the line up to a carriage return (which may be written explicitly as \n).

If ANGLE is non-zero, it will be used to determine the direction of the label, otherwise it is parallel to the x axis.

See the manual entry for LABEL or Appendix X for a description of how to enter a label with funny characters, sub- and super-scripts, and so forth.

If EXPAND is set to exactly 1, and ANGLE is exactly 0, then SM will use hardware fonts, when available, in writing labels. This is faster, but if you don't like it say "EXPAND 1.00001", or use a \r explicitly to select the roman font.

Syntax: YLABEL `str`

Write the label `str` centered to the left of the yaxis made by BOX. The string is taken to be the rest of the line up to a carriage return (which may be written explicitly as \n).

If ANGLE is non-zero, it will be used to determine the direction of the label, otherwise it is parallel to the y axis (ANGLE 360 will achieve horizontal labels).

See the manual entry for LABEL or Appendix X for a description of how to enter a label with funny characters, sub- and super-scripts, and so forth.

If EXPAND is set to exactly 1, and ANGLE is exactly 0, then SM will use hardware fonts, when available, in writing labels. This is faster, but if you don't like it say "EXPAND 1.00001", or use a \r explicitly to select the roman font.

Appendix I: How The Command Interpreter Works

The basis around which the command interpreter is written is a grammar which is passed a set of tokens (analogous to words in English) which it parses, given a set of grammatical rules. As it recognises each rule, it executes the code associated with that rule. The complete grammar for SM is given in Appendix IV, with the C code stripped out.

An example would be:

```
aa :  BB CC
   {
      printf(''Rule BB CC found\n'');
   }
```

which specifies that the rule aa consists of the token BB followed by CC, and that if rule aa is recognised the programme should print that fact out. Conventionally, uppercase names are reserved for 'terminal symbols', and lowercase for 'non-terminal symbols' where terminal symbols are those that are passed to the parser (analogous to words), and non-terminal symbols are tokens that the parser has constructed out of terminal symbols (analogous to phrases). The right hand side of a rule may contain a mixture and non-terminal symbols, and symbols may be assigned a value *.

Token Generation

SM generates tokens for the grammar roughly as follows: When characters are typed at the keyboard, they are read by a routine which runs in CBREAK mode (PASSALL for VMS), and receives each character as it is typed. It is this routine that handles command line editing, the history system, and key bindings. † Following a carriage return, it passes the whole line to the lexical analyser, which divides the input stream into integers, floats, strings, or words. In addition it recognises `${}^` as having special meanings (see below under variables ($) and history (^)). As in C, the escape sequence '\n' is replaced by a newline, which means that commands which read to the end of the line may be fooled into thinking that they have found it; see the examples at the end of the section. A { sets the flag 'noexpand', which turns off the interpretation of all special symbols, and causes all tokens to be returned as WORD. The matching } unsets this flag. This mechanism is used in defining macros and various lists. A word is anything which is not

* The grammar is actually specified using YACC, see S.C. Johnson *YACC: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, 1975, AT&T Bell laboratories, Murray Hill, NJ07974. This report is reprinted in section 2b of the UNIX manual, and is rather difficult reading at first. We do not in fact use the AT&T code, which is proprietary, but rather a public domain compiler-compiler called Bison written by the Free Software Foundation.

† Specifying -s on the command line bypasses all of this, and makes SM read input one line at a time.

otherwise recognised, so for example 'hello_there.c' or '1.2e' would be considered words. Symbols are separated by white space, taken to be spaces tabs or newlines, or the characters '!', '{', '}', '+', '-', '*', '/', '=', '?', '!', ',', '<', '>', '(', or ')'. This behaviour can be modified by enclosing a string in double quotes, when no characters (except ^) are special, and tokens are delimited only by the end of the line, or some character after the closing quote. Enclosing in quotes is rather similar to enclosing in { } , except that quotes have no grammatical significance. A string in double quotes is always treated as a word, but the quotes must not have been discarded by the time that the lexical analysis occurs. For example, "2.80" is a float, as SM will have digested the " before looking at the string. You can fool it with "2.80 ". A string begins with a ' and continues to the next ': they are used in certain contexts where SM needs to know if a **WORD** or **STRING** is involved, for example in a **PRINT** command. It's worth noting that the '...' are stripped when the string is recognised – if you need to preserve them make sure that **noexpand** is set (*e.g.* **SET** s={ 'a' 'b' 'c'}).

The output from this programme is passed to a second stage of lexical analysis. This passes integers and floats through unaltered, while words are passed through a filter to see if they are external tokens from the grammar (such as **CONNECT**). If a word is recognised as being a token then that token is returned, otherwise the token **WORD** is passed, and the text of the word is stored. Tokens may be written in either lower or upper case, but for clarity they are written in upper case in this document. The overloading of lowercase tokens is achieved at this stage by simply refusing to recognise them as keywords.

The input stream is now fully analysed into tokens and is passed to the parser, which is written in YACC. If the sequence of tokens seen corresponds to a grammar rule, the parser executes the appropriate section of code, which is written in C. If the parser doesn't understand, it tells you that you have a syntax error and prints the last logical line that it was processing, with the error underlined. If you can't figure out which command it really failed on, try setting the **VERBOSE** flag to be 4 or more. This produces a voluminous output, which will stop suddenly when the error re-occurs. One simple rule in the grammar is that a **WORD** should be treated as a possible macro.

Peculiarities of the Grammar

If the command interpreter is faced with a pair of grammar rules such as

```
AA BB CC
```

and

```
AA BB
```

it may not know whether to treat the tokens **AA BB** as the first part of **AA BB CC** or as the complete command **AA BB** followed by the token **CC** beginning the next command without examining the next token. This ambiguity only arises if a command can begin **CC**, and may be dealt with by defining the second rule as

```
AA BB '\n'
```

This should be borne in mind whenever SM complains about a syntax error in an apparently valid command (such as `LIST MACRO HELP`, intended as first `LIST MACRO` and then the valid command `HELP`). The presence of a required carriage return also sometimes requires that macros be spread over a number of lines rather than as one long list of commands, although a carriage return may always be written as '\n', which makes SM think that it has found a carriage return. There is a also requirement that an `ELSE`less `IF` statement should end with a newline; this is produced by a subtlety of the way that `IF`'s are processed and is discussed under `IF`.

SM places a restriction upon commands such as `RELOCATE` which expect more than one argument, which is that the arguments must be numbers rather than (scalar) expressions. This is required by the unary minus, as if the grammar sees `expr1 - expr2` it cannot know whether this is the two expressions `expr1` and `-expr2`, or the single expression `expr1-expr2`. Unless the grammar is changed, for instance by using commas to separate arguments, this restriction cannot be lifted; it can, however, frequently be circumvented using macros such as `rel` discussed under 'Useful Macros'. As an alternative, in almost all cases the expression can be enclosed in parentheses, for example `connect (lg(x)) (-lg(rho))`.

The Macro Processor

Executing a macro consists of substituting the text of the macro for its name. In order to understand how SM does this you have to know a bit more about how it processes input characters. We said above that it 'passed the whole line' to the lexical analyser. What it actually does is to pass a pointer to the line, and starts reading from the beginning of the line. Now if you execute a macro, all that is done is that we now pass a pointer to the text of the macro, and start reading from it instead. The old pointer is pushed onto the top of a stack. When SM comes to the '\0' at the end of the macro text, the stack is popped and input continues as if the macro had never been seen. When we come to the end of the 'whole line' pushed at the top of this paragraph, *it* is popped, and SM gives you a prompt for more input. Of course, if a macro had been seen while the first macro was being executed, the first one would get pushed onto the stack, and attention transferred to the the new one. If a macro has any arguments, their definitions are pushed onto an argument stack which is popped at the proper times. To jump ahead a little, variables are implemented in a very similar way, being pushed onto the stack, as are `DO` and `FOREACH` loops, and perhaps more surprisingly `IF` statements.

The strange behaviour of `RETURN` at the end of macros comes about because when the input routine is reading the `RETURN` it has to read one character beyond it, so as to know that it isn't dealing with, say, `RETURN_OF_THE_NATIVE`. But in looking for the next character it has to pop the macro off the stack, so when the `RETURN` is acted upon we have *already* returned from where we wanted to return from, and we now `RETURN` from the wrong place. In a similar way, an `IF` at the end of a macro will cause the parser to look for an `ELSE`, thereby popping the macro stack if there isn't one. If the `IF` test was true, and contained references to macro arguments, there

will be a problem as either there will be no macros defined, or the arguments to the previous macro on the stack will be supplied.

Macro definitions are currently stored in the form of a weight-balanced tree (actually a BB($1 - \sqrt{2}/2$) tree). This means that the access time for a given macro only grows as the logarithm of the total number defined. In the future it may be possible to choose the weights depending on the access probability for a given macro, but this is not currently possible. Definitions of variables and vectors are stored in the same way.

The DO, FOREACH, and IF commands

It seems worth discussing the implementation of these commands. Both loops consist of a definition of a variable, together with instructions about what to do with it, followed by a list of commands within a set of { } , while `IF` just has the command list. It is not possible for the main grammar to execute commands or macros, as the YACC implementation is non-reëntrant, so the best that it can do is to push the commands onto the input stack as a sort of temporary macro, after defining the initial value of the loop parameter. When the '\0' at the end of the loop appears, instead of popping the macro stack we simply define the loop parameter to have its next value, and jump back to the beginning. This means that you can't change the value of a loop parameter, as it'll be reset anyway, but you can use it as a sort of local variable.

`IF` statements are similar, in that we read the entire list before executing it. Once more, a temporary buffer is pushed onto the stack, with instructions to delete it after use. The reason that a newline is required after an `ELSE`less `IF` is that the grammar will have already read the next token to see if it was `ELSE`. If it wasn't, then it will seem to have been typed before the body of the `IF`. For example, `IF( test ) { echo Hello } PROMPT :` will be parsed as `IF( test ) { PROMPT echo Hello } :` if test is true, but correctly as `IF( test ) { echo Hello } PROMPT :` if it is false. Because an extra \n does no harm, we demand it.

Examples

If you want to watch SM thinking about these examples, the command `VERBOSE 4` will make it print out in detail each token as it reads it, and each macro or variable as it expands it. To turn this off, use `VERBOSE 0` or `VERBOSE 1`. To really see the parser at work, try a negative value of verbosity. This will report every step that the parser takes, providing that it was compiled with DEBUG defined. A second negative value will turn the information off again.

`PROMPT @` — `PROMPT` is an external token, so `PROMPT` is passed to the grammar which recognises the rule `PROMPT WORD`, and sets the prompt to be '@'. When it has finished, control is passed back to the input routine.

```
MACRO p { PROMPT }
```

This is a simple macro defining p to be PROMPT

```
p @
```

The lex analyser doesn't recognise p as a keyword, so it returns WORD and as the grammar has no other interpretation of a WORD in this context, it passes p to the macro interpreter, which replaces it by PROMPT (*i.e.* pushes PROMPT onto the input stack). SM now thinks that you have just typed PROMPT @, and behaves as described in the first example.

```
MACRO pp 1 { PROMPT $1 }
```

The macro pp is declared to have one argument, which is referred to as $1. After pp is invoked it reads the next (whitespace delimited) word from the input stream, and replaces $1 by that word.

```
pp @
```

Just like the first example, the prompt is set to @.

```
pp
```

You are prompted for the missing argument to PROMPT.

```
PRMPT
```

As PRMPT isn't an external token, it is a WORD, so SM tries to execute it as a macro and complains if it isn't defined.

```
DEFINE Hi Hello
```

The variable Hi is defined to have the value Hello.

```
WRITE STANDARD $Hi Gentle User
```

When it has read $Hi SM pushes the value of the variable Hi onto the stack and then reads it, popping it off again when it has finished. The WRITE STANDARD command writes Hello Gentle Reader (*i.e.* up to the end of the line) to the terminal.

```
WRITE STANDARD $Hi Gentle User \npp "SM>"
```

As above, the rest of the line is written to the terminal (up to the carriage return '\n'), then the prompt is changed yet again.

Appendix II. The Stdgraph Graphics Kernel

SM can use a single set of subroutine calls to plot on almost any terminal, and on many printers. The routines that it uses, called stdgraph, were taken from the IRAF GIO package written at Kitt Peak by Doug Tody * and converted to C and partially re-written to be integrated into SM. Despite our extensive rewrite, these routines should probably still be considered to be in the public domain.

Graphcap.

Stdgraph uses a file called a graphcap file to specify the properties of terminals, in a way that is similar to the termcap facility of Unix. You don't have to know anything about termcap to read this section; you don't have to read this section unless you want to change the graphcap file to add a new device, to fix a bug, or to change the way that SM treats your terminal. The name of the graphcap file is given by the variable `graphcap` in the environment file. A list of files to be searched in order may be given instead of a single graphcap file (up to a current maximum of three).

A graphcap file is a way of describing a terminal in a concise way, so a programme can discover which idiosyncrasies a terminal has without having to be recompiled. A graphcap file consists of a number of entries, one for each device supported, and to add a new terminal all that one has to do is to add another entry. It is possible to compile selected entries in the graphcap file, so as to improve access time for popular terminals. If this has been done, changing the graphcap file for one of these terminals will have no effect until it is recompiled. See the section on 'compiling graphcap' for details.

Each entry consists of a name for the device, followed by a list of aliases, followed by a list of fields, separated by colons. A \ may be used to continue an entry onto the next line, and lines starting with a # are comments (comment lines are only permitted between entries). As a rather complex example, the graphcap entry for a Tektronix 4012 reads:

```
tek4010|tek4012|TEK4010|TEK4012|Tektronix 4010/2:\
    :ch#.0294:cw#.0125:co#80:li#35:xr#1024:yr#800:\
    :MC=^M:CL=^[^L:CN#6:GD=^X:GE=^[1^]:\
    :ML=^[(1$0)'($1)a($2)c($3)d($4)b($$:lt=01234:\
    :OW=^]^_:RC=^[^Z:SC=(,!3, & *, &+!1, & *, &+!2:\
    :TB=^]%t^_:VS=^]:\
    :xr#1024:XY=%t:yr#780:
```

This is one of the longest entries in the graphcap file - all of the terminals which are Tektronix emulators explicitly include this entry, so they only need provide the capabilities that are different from the Tektronix. As an example, the entry for a Pericom reads

* Graphics I/O Design, Doug Tody, March 1985. NOAO (Kitt Peak)

```
pericom|Pericom:\
    :GE=^]:TB=^](2#7-!2)%t^_:\
    :tc=tek4012:
```

The | separate the aliases, and the final field `tc=tek4012` tells stdgraph to take all other fields from the entry for `tek4012`, given above. If you have specified a list of graphcap files, each will be searched in order for each `:tc=` continuation.

Control characters are entered as ^A, ^B, and so on (those are two characters, ^ and 'A'). 'Escape' may be represented as ^[, \E, or in octal as \033. Because the normal way of handling strings in C treats \0 as meaning 'end of string' you can't simply put a \000 into a graphcap entry, instead write \377 and SM'll interpret it as \ . (If you need a real \377 enter \377\377). If a delay of so many milliseconds is required before the transmission of a string, it is given first (followed by a * if it is to be applied to each line affected). Numerical values are preceded by a #, so `:co#80:` means that co (the number of columns displayed) is 80, while `:MC=^M:` means that MC (the cursor delimiter) consists of the character ^M. This could just as well have been written `:MC=\010:`. If the first character of a capability is '@', it specifies that that capability is not present for that terminal (*e.g.* `:lt@=1234:` specifies that `lt` is not defined). A field may simply not be provided if it is irrelevant, although in this case it may be supplied by a `tc` continuation. A common set of graphcap entries to 'comment out' are `TB` and `TE`, which deal with hardware character sets. If you don't want your plotter to use it's internal fonts simply insert '@' before the '='. By inserting their private file before the system one in the list of graphcap files, users can tailor the entries to their liking.

We use a subset of the graphcap capabilities defined by the IRAF group, and the distinction between upper and lower case parameters comes from them. In a few cases our usage is different from theirs, in these cases we have specified our own capabilities (`CD` → `MC`, `DD` → `SY`, `LT` → `ML`, and `TS` → `TB`. We have also added the `lt`, `BP`, `BR`, `CO`, `CS`, `CT`, `DC`, `DT`, `EP`, `ER`, and `RA` capabilities.). First the lower case, which specify mostly dimensions:

ch	Height of a character, relative to the screen height being 1.0.
co	Number of columns displayable, with characters of width cw.
cw	Width of a character, relative to the screen width being 1.0.
li	Number of lines displayable, with characters of height ch.
lt	Which linetypes are supported in hardware.
pc	Pad character for delays (use NUL if not supplied).
xr	Maximum number of points plotted in x.
yr	Maximum number of points plotted in y.

Of these, `co` and `li` are not currently used.

The capitalised capabilities mostly tell the stdgraph routines how to plot lines, clear the screen and so forth. Some of these are no more than character strings to send to the terminal, (e.g. CL to clear a screen), but some use the graphcap entries to programme a sort of RPN calculator, which computes the bit-patterns that the terminals demand. This calculator is usually referred to as the 'encoder'. We'll first

list all the capabilities in a reasonably ordered way, then describe the encoder and what it can do, and then go through a number of examples.

First the fields which are simple character strings to be written to the terminal. The second column is an attempt to explain the etymology of the two character name.

CL	CLear	Clear the screen, possibly also the text screen.
CW	Close Workstation	Close terminal, expect no more graphics.
DS	Draw Start	Prepare the terminal to draw a line.
DE	Draw End	Finish a line.
FD	Fill Draw	Draw a side of a filled polygon.
FE	Fill End	Finish drawing a filled region.
FS	Fill Start	Start drawing a filled region.
GD	Graphics Disable	Return the terminal to a character mode.
GE	Graphics Enable	Set the terminal to graphics mode.
IF	Initialisation File	Used to supplement OW if sequence is too long.
LR	Load Registers	(Used by the RPN encoder, see 'binary encoding').
ME	Mark End	Finish a series of dots movements.
MS	Mark Start	Start a sequence of dots movements.
OW	Open Workstation	Prepare a terminal to produce plots.
VE	? End	Finish a series of pen (beam) movements.
VS	? Start	Start a sequence of pen movements.

The GD and GE are used by terminals which spend some of their time being graphics terminals, and some being regular text terminals. The various "... Start" and "... End" capabilities assume that the points in question are specified by the XY entry (except for FS/FE where FD is used instead). Typically, the 'start' is used to put the device into (*e.g.*) line-drawing mode, then the line is drawn with a sequence of XY's, then it is taken out of (*e.g.*) line mode with the 'end'. The support for filling areas assumes that a region is specified by drawing a line around it; if this isn't so, you'll have to omit area fill from graphcap, and rely on SM emulating it for you. An example would be a Graphon GO-250, which has an area fill where you fill rectangular areas by specifying opposing corners; this is not acceptable to SM.

Some operations require an argument, for instance setting the hardware line type, specifying which cursor to read*, or specifying coordinates. In the following properties, the expected parameters are listed after the field names, the first to go into register 1, the second into register 2, and so on. If you haven't skipped forward

* Actually, SM always uses cursor 1

to the section on the encoder this will seem obscure, but all will become clearer.

CO(r,g,b)	COlour	Set next colour to (r,g,b).
CS(n)	Colour Start	Start defining n colours.
CT(i)	Colour Type	Set a colour.
DC	Default Colour	Set default colour.
MC(i,x,y)	sM Cursor	Decode a cursor response.
ML(i)	sM Line	Set the linetype to i.
RC(c)	Read Cursor	Read the cursor. 'c' is for compatibility.
SC	Scan Cursor	Decode the cursor reply following a RC.
TB(x,y)	Text Begin	Start writing text at (x,y).
TE	Text End	Stop interpreting characters as text.
XY(x,y)	X Y	Encode the coordinate pair (x,y).

Some of the above comments are a little cryptic, but we return to the various graphcap parameters that take arguments as examples after describing the encoder. Note that it isn't sufficient to change the `ML` entry — for a linetype to be supported in hardware it must also be included in the `lt` list, e.g. `lt=01234`. Similarly, for hardware fonts you must include `ch` and `cw`, and `TB` must be present even if it does nothing.

The folowing capabilities have to do with rasterising and are discussed in their own section near the bottom of this appendix:

BP	Bit Pattern	Bit patterns for rastered data.
BR(i)	Begin Row	Begin row of rastered data.
EP	Empty Pattern	Bit pattern for rastered empty pixel.
ER	End Row	End of a row in rastered data.
MR	Many Rows	Number of rows output at once.
nb	nUM bYTES	Number of bytes to process at once for MR.
RA	RAster	This device is a raster device.

Raster devices also make use of `xr`, `yr`, `CW`, `OW`, `OF`, and `SY` which are also used by stdgraph itself.

Finally there are three capabilities that are designed for driving hardcopy devices.

DT	Device Type	Type of device in use.
OF	Out File	The file to direct output to.
RT	Record Terminator	(RT is no longer supported).
SY	SYstem	The action to be taken upon closing the OF file.

The OF file may be specified with the last characters being 'XXXXXX', in this case the Xs are replaced by a random characters, to make a unique filename. If the variable `temp_dir` is defined in the environment file, then OF is created in that directory, otherwise it is put in the current directory. The DT string, if present, specifies the type of device in use. Currently the values are only used under VMS, where they are used to decide how to open files. The recognised values are "qms"

and "imagen". In general DT should be omitted, as it requires programming support, but it can help stdgraph to deal with hostile operating systems.

The SY string is passed to the operating system after any occurrences of '$F' have been replaced by the filename specified by OF, and '$n' has been replaced by the n^{th} argument to the DEVICE command. A $ may be escaped with a \, but the \ must itself be escaped so to include a dollar in an SY string, type \\$. This means that (under Unix) you can access environment variables from `SY` strings, *e.g.* `:SY=\$SM/rasterise -r $0 $F -`. For example, if the DEVICE command was `DEVICE stdgraph qms lca0 Hello` (or `DEVICE 1 qms lca0 Hello` or `DEVICE qms lca0 Hello`), then the device name `qms` would be $0, `lca0` would be $1 and `Hello` $2. The SY string is only used if an OF file has been specified. There is no guarantee that SY is supported by all operating systems, but it is certainly available under Unix and VMS (SY requires the C call 'system()', as defined for Unix. We have provided one for VMS, and any serious SM implementation would have to have one too.) A trivial example of SY in use on an Unix system would be:

```
:SY=cat $F ; rm $F:OF=out_XXXXXX:
```

(cat prints a file, ; separates multiple commands on a line, rm deletes a file). Because not all operating systems can support multiple commands on one line, you can use \n within a SY string to separate commands. For example, under VMS that `SY` string could have been written

```
:SY=type $F. \n delete $F..*:OF=out_XXXXXX:
```

(Type adds a '.lis' unless explicitly given a closing '.', delete requires a version number, hence the `$F.` and `$F..*`.)

The `RT` capability has been deleted in version 2.0, in favour of using `DT`.

The Binary Encoder.

Different terminals have very different ways of doing the same thing. For example to move the beam to (200,200), a vt240 in REGIS mode needs to be told '[200,259]', while a Tektronix 4010 needs '&h&H'. In order to cope with this much diversity, stdgraph has a binary encoder with a 50 element stack, 10 registers and about a dozen operators. The encoder communicates with the rest of the world through its registers - for example in encoding a coordinate pair it expects to find x in register 1, and y in register 2. When reading a graphcap string, initially stdgraph simply copies the input characters to an output string, which is then written to the terminal. This is exactly what it does when it interprets the OW string for a Tektronix, `OW=^]^_`. However, in addition to characters such as ^ being special, it also recognises the following as being special:

'	escape special meaning of next character
%	Begin a format string
(	Switch from copy into encode mode.

When in 'encode' mode, the following operators are available:

'	Escape next character (recognised everywhere)
%	Formatted output, e.g. %d or %t
)	Revert to copy mode
#nnn	Push signed decimal number nnn onto the stack
$	Part of a switch statement
.	Pop a number from the stack, and put it in the output string
,	Get next number from input string, and push it onto the stack
'str'	Prompt with str, then read a character and push it onto the stack
&	Modulus operator (similar to an AND of the low order bits)
+	Add (similar to OR)
-	Subtract (similar to AND)
*	Multiply (a left shift if number is a power of 2)
/	Divide (a right shift if number is a power of 2)
<	Less than (0:false, 1:true)
>	Greater than (0:false, 1:true)
=	Equals (0:false, 1:true)
;	Branch, <boolean><offset>; (; is at 0 offset)
0-9	Push register 0-9 onto the stack
!N	Pop the bottom of the stack into register N.
!!	Pop the stack, and delay that many milliseconds.

All the binary operators operate on the bottom 2 elements of the stack, and push the answer onto the bottom. Any other character is interpreted as an integer, and pushed onto the stack - for instance, '@' is the same as '#64', octal 100. A blank is the octal constant 040.

The % command means, 'format the bottom of the stack, and write it to the output string'. The format string may be any printf format specifier (printf is the C formatted i/o function. In practice, the only formats that you are likely to need are %c, %d, and %t – and %t isn't even in C! %c means 'write the integer as a character', %d means 'format the number as a decimal integer', and %6d means 'and make it fill 6 characters'. If you should need to know more, look at any book on C.) The special format %t means 'take x and y from registers 1 and 2, and format them for a Tektronix'. As we shall see below, you can programme the encoder to do this, but Tektronix emulators are so common that %t is provided for efficiency's sake. In fact there are two Tektronics formats, %t for 10 bit addresses, and %T for 12 bit addresses. The switch and branch instructions are discussed below, while examining specimen `ML` and `SC` strings.

Examples

As a simple example, the ANSI command to set a non-graphics cursor to a given line and column is

^[[line ; column H

Assuming that the x and y coordinates are in registers 1 and 2 respectively, the corresponding graphcap string would be

```
"^[[(2)%d;(1)%dH"
```

(where the quotes are not part of the format.) What if line and column coordinates start at 1, but the terminal wants them starting at 0? then the format would be

```
"^[[(2#1-)%d;(1#1-)%dH"
```

As promised above, it is also possible to encode Tektronix-type coordinates. The desired bit format for a 10-bit address is

0 1 ya y9 y8 y7 y6
1 1 y5 y4 y3 y2 y1
0 1 xa x9 x8 x7 x6
1 0 x5 x4 x3 x2 x1

where x1 is the least significant bit in x, and ya is the tenth bit in y. If x and y are in registers 1 and 2, the simplest XY (move/draw to (x,y)) string is

```
"%t"
```

but if this weren't available the following string would work:

```
"(2 / +.2 &'+.1 / +.1 &@+."
```

(as before, the double quotes don't belong to the format). To understand this, First look up the octal values of ' ' (040), "'" (0140), and '@' (0100). Then the first `'('` puts the encoder into encode mode. `'2 /'` pushes the Y value onto the stack, and right shifts it by 5 bits (' ' is 100000 in binary). The next `' +.'` adds the resulting bit pattern '0 0 ya y9 y8 y7 y6' to 0100000 and transfers it to the output string, and we have produced the desired first byte. The other bytes are produced in a similar fashion.

As another example consider an AED512, which is reputed to desire the bit sequence

xa x9 x8 yb ya y9 y8
x7 x6 x5 x4 x3 x2 x1
y7 y6 y5 y4 y3 y2 y1

The graphcap string

```
"(#128!919/^N*29/+.19&.29&."
```

will accomplish this. We could further optimise this by loading the value '#128' into register 9 once and for all with the LR capability, so a part of the graphcap entry would appear as

```
":LR=#128!9:XY=(19/^N*29/+.19&.29&.:"
```

I've never seen an AED512, but this should work anyway.

The switch instruction has the form

```
$i ... $j-k ... $l ... $D ...
```

where `i`, `j`, `k`, and `l` are integers. The encoder pops the bottom value off the stack adds '0' to make it a character, and scans forward looking for a `$` followed by that character. `$2-5` would match the characters '2', '3', '4', or '5'. When it has met its match, it executes the instructions that it meets until it reaches the next `$` in execute mode. The encoder then skips forward until just after the `$$`, and resumes

scanning. If the character from the stack is not matched by any of the cases, the encoder will use the `$D` (*i.e.* default) case, if present.

As an example, consider how stdgraph sets the type of line to draw. SM expects linetype 0 to be solid, 1 to be dotted, and so on. We expect a linetype in register 1 and have to do something with it.

For a Tektronix, the linetypes are set by an ML entry:

```
ML=^[(1$0)'($1)a($2)c($3)d($4)b($$
```

What does this do? The `^[` is simple, it is executed in copy mode, and writes the character `^[` to the output string. The `(1` enters encode mode, and places the contents of register 1, the desired linetype, on the stack. Then begins the switch. If the linetype is 0, then the encoder scans past the `$0` and starts reading the string again with `)'`. The `)` takes the encoder back to copy mode, so it copies `'` to the output string, and encounters a `($` which puts it back into encode mode. Once in encode mode it recognises the `$` as the end-of-case, and scans forward until it reaches `$$`, where it stops. We deduce that the set-linetype-0 escape sequence is `^['`. If register 1 had contained a 2, after entering the switch the encoder would have scanned forward to `$2` (ignoring all characters as it went), and copied `c` to the output string.

If you want to support erasing of individual lines (`LTYPE ERASE` or `LTYPE 10`) you'll have to include a `$\:` case in your switch (as `:` follows `9` in the ascii character set, and an un-escaped `:` would end the graphcap entry). You'll have to escape the `:` in the `lt` list as well. When leaving erase mode, by specifying any other line type, the device will first be set to `LTYPE 11` (*i.e.* `ML`'ll get a `;`) before it's set to the desired `LTYPE`; this gives the driver a chance to reset itself. It's wise to also turn off erase mode when closing the device. An example of an entry supporting erasing lines is a graphon, which includes

```
:lt=01234\:;:CW=^[1^]^[^A^[2\
:ML=^_^[(1$0)'($1)a($2)c($3)d($4)b($\:)'^[^P($;)^[^A($$:
```

as `^[^P` puts a graphon into erase mode, and `^[^A` takes it out. Note that in erase mode the linetype is set to solid (`^['`), so as to erase all types of lines.

There is also a branch instruction, which has syntax

```
<boolean><offset>;
```

If the boolean is true (non-zero), then skip (offset - 1) characters in the programme string. The offset may be either positive or negative, and the ';' is at offset 0. For example,

```
(0#15;)Goodbye(#1#8;)Hello()\n
```

will print 'Goodbye\n' if register 0 contains zero, or 'Hello\n' otherwise. As an example of the use of ';', consider using the encoder to decode a string. Remember that ',' meant 'read a character onto the stack', and that there was a graphcap capability SC to decode cursor responses. Suppose that we are dealing with a vt240 in REGIS mode, then a cursor read will return a string of the form 'k[nnn,mmm]' where 'k' is the character you hit, and (nnn,mmm) is the cursor position. We want to put k into register 3, and (x,y) into registers 1 and 2. This is a little messy, as we'll have to convert the ascii positions into integers. The desired graphcap entry

is

```
SC=(#0!1#0!2,!3,#0!8,#48-!99$0-91#10*9+!1#1!8$$8#1=#-39;\
    0!8,#48-!99$0-92#10*9+!2#1!8$$8#1=#-39;62-!2):
```

The first part is simple enough, store 0 in registers 1 and 2, store the first character in register 3, read a character (the [), and store 0 in register 8. Then we come to `,#48-!99$0-91#10*9+!1#1!8$$8#1=#-39;`. The `,#48-` reads a character and converts it to an digit (48 is the decimal code for '0'), then stores it in register 9. The switch then checks if we do have a digit, if so we multiply register 1 by 10 and add the new digit. We then set register 8 to 1 and finish the switch which is here being used as an if statement. The `8#1=#-39;` tests register 8 against 1 (*i.e.* checks if we found a digit), and if we did it jumps back 39 characters, to read the next character*. So we are accumulating the integer nnn in register 1, just as we needed to. The rest of the string deals with decoding the y coordinate.

Sometimes you don't want to read from the input string, but from the keyboard instead. In this case use `'str'`, *e.g.* `('Hello\: '#48-$0)False($D)True($$)\n:` will prompt you with `Hello:` , then read a character from the keyboard. If you enter a '0' it'll print `False`, otherwise it'll print `True`. Of course, in reality you'd want to do something more useful (such as erasing the screen).

Cursors

We have just been through a long explanation of how to decode a cursor string, but how did stdgraph know what to read in the first place? After receiving the RC string, the terminal will send back a sequence of bytes, and the format of these bytes must be specified in graphcap.† There are two ways to do this, either by specifying a sequence of characters which **end** the response string along with a minimum number of characters to read, or by specifying a pattern that the terminal response is to match. A typical example of the former is a Tektronix whose cursor response may be chosen to be `^M` (this is called the GIN response, and can usually be set in the terminal setup). We know that the terminal will also send 5 other bytes (the key struck and the encoded x,y coordinates so we would specify

```
MC=^M:CN=6:
```

On the other hand, a REGIS terminal sends 'k[nnn,mmm]'. This can be specified as

```
MC=?[#*,#*]:CN=-6:
```

where the negative value of CN means that we are providing a pattern not just a terminator (as before, the absolute value of CN is the minimum number of bytes in a cursor response). In MC strings, but nowhere else, the characters ?,#, and * are special (although their special meanings may be escaped with a \). ? will match

* In counting characters for jumps, the ; is at character 0 and combinations such as ^N count as one character

† If the RC string is given as `prompt`, then you will be prompted for the key you would have hit, and the (x,y) position the cursor would have been at, if the terminal that you were using could support a cursor.

any character, # any digit, and * means 'match zero or more of the preceding characters'. So a MC string of `a#*?ba` will match 'aaa1111bbaa' at the third character. (Incidently, `a#*?a` would match at the first). Because this special character syntax is different from that used in standard graphcap files for IRAF, the name of this graphcap parameter has been changed from `CD` to `MC`.

If your cursor is atached to a mouse, if possible the buttons should be set up to generate 'e', 'p', and 'q' from left to right (if you have that many buttons). If you have only one button, 'p' is probably the best choice.

Colours

The number passed to `CT` are the same as those specified with the `CTYPE INTEGER` command, so initially they specify default, white, black, blue, red, green, magenta, yellow, and cyan (white is 1). These are the colours corresponding to turning one, zero, two, or three of the primary colours on. The default colour to use for a device is specified by the `DC` capability, *e.g.* `:DC="red":`.

The `CS` and `CO` capabilities are used to support the `CTYPE = expr` command. First `CS` is used to tell the device how many colours to expect, then `CO` is used for each number, with red, green, and blue as its arguments. In this case `CT` passes an index into the set of `CO` values. If you want to get an index, but don't need `CS` and `CO`, you must still provide them; just provide a no-op such as `:CS=():`.

Writing a New Graphcap Entry

So, if you're faced with a new piece of hardware what should you do? First of all, don't panic – writing entries is quite simple. Second, see if your device is basically the same as one that already exists in graphcap, for example the entry for 'graphon' uses the 'selanar' entry, and it in turn uses 'tek4010'. You might be able to get away with using `tc` to satisfy most of your device's needs.

But let's assume that you are faced with a totally new type of device and really do have to start from scratch. First find out how large your device is, and fill in the `xr` and `yr` entries. If you are going to use hardware character sets you also need `ch` and `cw`. Next decide on the string to initialise the device – does it need to be set into some weird mode – and put it into `OW`. Put the string to reset it into `CW`. Now, if the initialised device needs to be put into a special graphics mode put it into `GE` and its inverse into `GD`. Next, you need to tell SM how to draw a line and move the plot pointer. So enter the `DS`, `XY`, `DE`, `VS`, and `VE` capabilities. Of course, if one isn't required, don't put it in. If you have some sort of printer you probably want to store all the commands in a file (`OF=`), and to plot them (`SY=`). You should now be ready to make your first test, so plot a box. If it doesn't look right, fix it. Or you might like to try printing the cover (`load cover cover`).

When all is well, you can begin looking into options that might make your graphcap entry more efficient. Look through this Appendix to see what is available. Does your device support line types? Add ML and lt. Heavy lines? LW. Coloured lines or a cursor? read the section on colours or cursors in this Appendix. Filled polygons? FS, FD, and FE. Dots? MS ME. Hardware characters? TS TE. When you have finished please send us your new entry.

Raster Devices

Stdgraph can only handle devices that can plot vectors specified by their endpoints; unfortunately some devices (such as most line printers) can only plot graphs when they have been reduced to rows of 'on' and 'off' pixels. SM supports such devices through DEVICE raster and a separate programme called rasterise. It communicates with the rasteriser through graphcap, so the whole process is user transparent. A separate rasterising programme was written so as to allow the plot to be produced in the background while you do more productive things, and to allow the rasterising to be done on a remote machine.

DEVICE raster produces a file, whose name is specified as usual by the OF field in graphcap, containing the vectors to be plotted (as groups of four short integers) in device coordinates, where the size of the device is taken from xr and yr. When the device is closed, the command specified by SY are executed, and these will usually be of the form rasterise -r $0 $F outfile\n print_it outfile\n delete outfile where print_it is the proper way of actually getting a plot. Under Unix, the command might well be something like (rasterise -r $0 $F - | lpr -v -r -P$1)& dispensing with the temporary outfile. In the same way that you never really need to say DEVICE STDGRAPH tek4010, but use DEVICE tek4010 instead, you can say DEVICE printronix rather than DEVICE RASTER printronix, providing that the graphcap entry for printronix includes the RA capability.

What do these rasterise commands do? The command syntax is rasterise [-flags] device infile outfile, where the infile may be specified as '-' to use standard input (sys$input to VMS), where the outfile may be specifed as '-' to use standard output (sys$output to VMS). Possible flags are r to remove the infile after use, R to rotate the plot through 90°, and v for more verbose operation. Rasterise then reads the data in the infile, and produces a rasterised version, row by row, on the outfile. In order to do this, it looks in graphcap for an entry for device, and uses the xr, yr, OW and CW fields as usual. †

Let's first consider a simple, one-line-at-a-time device such as a line printer. Before writing each row to outfile, rasterise encodes the BR (Begin Row) capability, using the current row number as an argument, and encodes ER (End Row) at the end of the line. By default, it assumes that the raster device simply wants bits turned on where a dot is required, but this can be overridden using the BP and EP capabilities.

† In looking for the graphcap file, any environment file specified on the SM command line with the -f flag is ignored.

EP (Empty Pixel) specifies the bit pattern for a character to represent white space. In the simple case mentioned a moment ago, this would be simply NUL, with no bits on, but sometimes this doesn't suffice (see examples below). BP (Bit Pattern) is a string, giving the bit patterns required to turn on the various pixels. In the default case, BP could be specified as BP=\001\002\004\010\020\040\100\200, so \001 would turn on the first (rightmost) dot. Because there are eight characters given in the string, raster assumes that it can fit eight pixels into a single character. If you don't specify a BP this is what will be used.

Some other devices (*e.g.* Epson printers) choose to print several lines at a time, so a single byte transmitted to the device might print 8 lines, but only the first pixel of each line. Such devices are described to graphcap by being given the MR capability and a number nb which describes how many bytes deep the printing band is (if omitted nb defaults to 1). In this case, BP is used to describe which bits are turned on vertically rather than horizontally but everything is otherwise the same as for the simple case.

As an example, consider the HP laserjet. You'd specify it as DEVICE laserjet, and its Unix graphcap entry reads:

```
laserjet|HP laserjet (high resolution):\
    :RA:xr#1280:yr#640:CW=^[*rB:OW=^[*r1280^[*rA:BR=^[*b160W:\
    :OF=hp_XXXXXX:\
    :(SY=/usr/local/sm/rasterise -r $0 $F - > /dev/hp)&:
```

On opening the device, it gets the string ^[*r1280^[*rA, setting the resolution and raster mode. Then, at the beginning of each rastered line it gets ^[*b160W specifying that 160 bytes are coming its way, then finally ^[*rB to restore it to alpha mode. (It doesn't need to know which row it is on, so the BR string doesn't tell it, and the default BP and EP are fine). After the input file is read it is deleted, and the output file is sent to the standard output, whence it is redirected to the proper device, in this case directly rather than through a spooler.

A more complex example is a printronix printer, which encodes 6 pixels in each byte, and requires that bit 7 be turned on. It also needs an escape sequence at the end of each line. The corresponding graphcap entry is

```
printronix|DEC printronix printer:\
    :RA:xr#792:yr#792:CW=^L:OW=^L:BR=@:ER=^E^J:\
    :BP=\001\002\004\010\020\040:EP=\100:\
    :OF=pr_XXXXXX:\
    :SY=(/usr/local/sm/rasterise -r $0 $F - | rsh wombat lpr)&:
```

We use EP to turn on the seventh bit everywhere, as required, and specify only 6 values for BP, so only 6 dots will be packed into each character. The BR entry is empty, and ER provides the needed escape sequences at ends of lines. In this case SY sends the plot over a network to machine wombat.

Some devices are not able to simply accept a string of bytes with an occasional escape sequence. For example, a versatec needs to have the bit order changed,

or a simple screen plotter might want to write a * if a bit is set and a space otherwise. If this is the extent of your pathology, you can deal with it via the provided capabilities. (Fortunately adding a * onto a space makes a *, so you can use `:EP= :BP=*:` for the latter.) If you have a really bad device, it is possible to add new coded device drivers to **rasterise**. For the convenience of such devices, when **rasterise** is run it first compares **device** with the names of devices that it knows about, and if it finds it it calls a different set of routines to deal with the rows of data. Otherwise it proceeds as discussed in the previous paragraph. This behaviour is similar to that of the **DEVICE** command in using stdgraph if it doesn't recognise a device name. (**Rasterise** only looks to see if the beginning of the device name matches its internal list, so you can drive a number of similar devices from one driver. For example, if you have a device called a 'heffalump', and have hard coded it under that name, **rasterise** will use it for devices called **heffalump_huge** or **heffalump_horrible** in graphcap.

If you find that you *do* need to write routines for some device, don't be too disheartened. **Rasterise** will still do the book-keeping and rasterising for you, your work will be limited to a couple of output routines. If you need to know more, see the source for rasterise.

Compiling Graphcap

(This section is really for someone maintaining SM.) Rather than have stdgraph read the graphcap every time that it opens a terminal, it is possible to compile the capabilities of the more popular terminals into the executable. This is done by preparing an include file which initialises the appropriate arrays, using the programme 'compile_g' in the main directory. After this file (called cacheg.dat) has been prepared, files depending on it must be recompiled and SM must be relinked. The use of compile_g is pretty much self-explanatory, you give it a list of the terminals you want and it produces the cacheg.dat file. Problems arise, however, if you don't have a valid cacheg.dat file, as then you can't compile compile_g in the first case. Fortunately, it is possible to bootstrap a cacheg.dat file (by defining BOOTSTRAP to the C-preprocessor), and proceed from there.

When stdgraph attempts to use the compiled capabilities, it checks that the current graphcap file has exactly the same name as the one that cacheg.dat was compiled from, if it isn't then it reads the graphcap file anyway. This provides a mechanism for those without C compilers to change the graphcap entries of pre-compiled devices. If you have a list of graphcap files, the name of the first is checked against the name in the cacheg.dat file.

Appendix III. Calling SM from Programmes

The SM callable interface is different from that of Mongo and corresponds directly to the interactive version. Almost all of the commands available in SM can be called from either fortran or C, the exceptions being those concerned with the macro processor, variables, history, and vector manipulations. We assume that if you want to call graphics routines directly, then you are prepared to take responsibility for such things. In C (and probably pascal, modula, or ADA), the calls have the same names as the commands, so to set the limits say `limits(0.0,1.0,0.0,1.0);`. On VMS, if you are writing fortran, you must prepend an 'f' to the command - `call flimits(0.0,1.0,0.0,1.0)`. If you forget this 'f', your programmes will compile, but they **won't** work. (They'll give segmentation violations, most likely). *

If you used an ANSI compiler to link SM (or your guru did) then you will need to provide prototypes for the SM functions that you call if you use the C interface (Fortran knows nothing of such things). This can be done by including the file **sm_declare.h**.

In what follows, we will assume that you are not writing fortran under VMS, so we will omit the leading effs.

To use SM functions, you must link with appropriate libraries. You will always need to link with the three SM libraries, `libplotsub`, `libdevices`, and `libutils` *in that order*. Specifically, under Unix, you'll need to include the files **libplotsub.a**, **libdevices.a**, and **libutils.a** when you link (it's probably easier to use `-lplotsub -ldevices -lutils`, along with a `-Ldir` if needed). and under VMS you'll need **libplotsub.olb/lib**, **libdevices.olb/lib**, and **libutils.olb/lib**. I can't tell you where they'll be on your system. In addition, you may need to link (after `utils`) any libraries used by the devices that have been compiled into your version of SM. For example, if you use the X-windows driver, you'll need to link with the X-library (`-lX`). Consult a local guru in case of any trouble - the person who installed SM has had to work this out already.

A list of functions giving the calling sequence for all the available functions follows, but first an example. Note especially the use of `graphics` and `alpha` to set the terminal to graphics mode, and return to a normal terminal afterwards. We would recommend always calling these, if they do nothing (*e.g.* on a laser printer) they'll do no harm. A related function is `gflush()` which will update graphics on the screen, for instance with stdgraph where output is usually buffered.

```
      integer NXY
      parameter (NXY=20)
      integer i
      real x(NXY),y(NXY)
c
      do 1 i=1,NXY
```

* Under Unix the loader can often distinguish fortran from C, so you may *not* need the 'f' – `call limits(0.0,1.0,0.0,1.0)`

```
            x(i) = i
            y(i) = i*i
1           continue
c
            call device('hirez')
            call graphics
            call limits(-1.,22.,0.,500.)
            call ctype('red')
            call box(1,2,0,0)
            call ptype(40.,1)
            call points(x,y,NXY)
            call xlabel('X axis')
            call ylabel('Not x axis')
            call alpha
            end
```

Remember, if you were on a VMS machine then all those calls would start 'f' - `call fgraphics` and so forth. Note that `box` takes all of its four possible arguments, and that commands such as `points` add an argument to specify the number of points to plot. You must, of course, ensure that you use the correct type of variables, passing integers or reals as required (and as listed below). If you are using C, you must carefully distinguish between passing by value, and passing by address (which we only use when an array is expected, and for returning a cursor position).

The functions are as follows. For fuller definitions of the arguments look at the main description of the command. The only ones that are different are `conn` for `connect` (due to a collision with a system routine), and `curs`, `defvar`, and `plotsym` because they don't quite correspond to any interactive commands. The arguments are declared in fortran in this list. `real x(n)` means that `x` is an array of size n. 'Real' means single precision. (So in C, `character` → `char *`; `integer` → `int`; `real` → `float`; `real()` → `float *`. We deal with converting the calling conventions from one language to another, but note comments at the bottom of this table.)

`alpha`	`Set terminal back to a normal state`
`angle(a)`	`Set angle to a (real a)`
`axis(a1,a2,as,ab,` `ax,ay,al,il,ic)`	`Draw an axis. (real a1,a2,as,ab;` `integer ax,ay,al,il,ic)`
`box(x1,y1,x2,y2)`	`Draw a box - note 4 args. (integer x1,y2,x2,y2)`
`conn(x,y,n)`	`Connect a line. (real x(n),y(n); integer n)`
`ctype(c)`	`Set colour to c (character c)`
`curs(x,y,k)`	`Return position of cursor (real x,y; integer k)`
`defvar(str1,str2)`	`Select variable str1 to str2 (character str1, str2)`
`device(str)`	`Select device str (character str)`
`dot`	`Draw a dot`
`draw(x,y)`	`Draw to (x,y) in user coords (real x,y)`
`erase`	`Erase screen`
`errorbar(x,y,e,k,n)`	`Draw error bars (real x(n),y(n),e(n); integer k,n)`
`expand(e)`	`Set expand to e (real e)`
`format(xf,yf)`	`Set format strings to xf and yf (character xf,yf)`

```
gflush                  Flush graphics output
graphics                Set terminal into plotting mode
grelocate(x,y)          Relocate in device coordinates (integer x,y)
grid(i)                 Draw a grid (integer i)
hardcopy                Close current device, if appropriate make hardcopy
histogram(x,y,n)        Draw a histogram (real x(n),y(n); integer n)
identification(str)     Identification string (character str)
label(str)              Draw a string (character str)
limits(x1,x2,y1,y2)     Set limits (real x1,x2,y1,y2)
location(x1,x2,y1,y2)   Set location (integer x1,x2,y1,y2)
ltype(lt)               Set ltype (integer lt)
lweight(lw)             Set lweight (integer lw)
notation(xl,xh,yl,yh)   Set axis format defaults (real xl,xh,yl,yh)
plotsym(x,y,n,sym,ns)   Draw user points (real x(n),y(n);
                            integer n,sym(3*ns),ns)
points(x,y,n)           Draw points (real x(n),y(n); integer n)
ptype(pp,n)             Set point type - vector form (real pp(n); integer n)
putlabel(i,str)         Position a label (integer i; character str)
relocate(x,y)           Relocate in user coords (real x,y)
shade(delta,x,y,n,type) Shade a region (integer delta; real x(n),y(n)
                            integer n,type)
ticksize(xs,xb,ys,yb)   Set ticksize (real xs,xb,ys,yb)
window(nx,ny,x,y)       Select a window (integer nx,ny,x,y)
xlabel(str)             Draw x-axis label (character str)
ylabel(str)             Draw y-axis label (character str)
```

`Curs` returns after you hit any key, returning the coordinates of the point, and the key struck as an integer (*e.g.* 'a' as 97 in ascii). In C, all three arguments are pointers, two to float and one to int. `Defvar` defines a variable, it is identical to the interactive command `DEFINE name value`. It is only used to define variables that are significant to SM, currently `file_type` and `TeX_strings`. `Plotsym` is like first using the `PTYPE { ... }` command to define `sym`, and then `POINTS` to plot x against y. The `ns` array consists of triples of integers, `(move x y)` where move is 1 to move the plot pointer, 0 otherwise. This is not quite the same as the interactive command, but doesn't involve any characters. The `move` integer may not be omitted. Shade shades the inside of the curve specified by x and y if type is 1, or the area below a histogram specified by x and y if type is 2. The line spacing in screen coordinates is delta.

The use of the 2-D functions may require a little more explanation. If you want to use SM to read your data, producing contour plots is very similar to doing so interactively, with the difference that instead of defining the variable `file_type`, you must call the function `filetype` with the desired value as the argument before reading the data. This is equivalent to calling `defvar` to define the variable `file_type`, and is only supported for backwards compatibility. As an alternative, you can fill your own data array, and pass it to SM to be contoured with `defimage`. The function calls follow (again in fortran), followed by some explanation.

```
contour                 Contour current 2-D image
defimage(arr,x1,x2,y1,  Define an image (real arr(nx,ny),x1,x2,y1,y2;
```

```
    y2,nx,ny)                    integer nx,ny)
delimage                     delete current 2-D image
filetype(type)               Set 2-D filetype (character type)
levels(l,n)                  Set levels for contour (real l(n); integer n)
minmax(x,y)                  Get minimum and maximum of image (real x,y)
readimage(file,x1,x2,        Read a 2-D image (character file;
    y1,y2)                       real x1,x2,y1,y2)
```

The translation to other languages is not quite as simple as above. In C, **minmax** expects to be passed pointers to floats, and the first argument to **defimage** is not a 2-D array, but an array of pointers to the rows of the image. **x1, x2, y1, y2** specify the limits as in the interactive **IMAGE** command; if they are set to be 0.0 then the dimensions of the array will be used. As an example, again in non-VMS fortran:

```
      integer NXY
      parameter (NXY=20)
      integer i,j
      real z(NXY,NXY),lev(NXY)
c
      do 2 i=1,NXY
          do 1 i=1,NXY
              x(i,j) = (i-NXY/2)**2 + (j-1)**2
1     continue
2     continue
c
      call device('postscript latypus')
      call graphics
      call limits(-1.,21.,-1.,21.)
      call box(1,2,0,0)
      call defimage(z,0.,20.,0.,20.,NXY,NXY)
      call minmax(amin,amax)
      do i=1,NXY
          lev(i)=amin + (i - 1.)/(NXY - 1)*(amax - amin)
3     continue
      call levels(lev,NXY)
      call contour
      call hardcopy
      call alpha
      end
```

Appendix IV. The SM Grammar

This is a specification of the grammar used to describe the SM command interpreter, written in YACC but with all the C-code removed. This may help you in understanding otherwise bizarre objections to your favoured command, as it specifies exactly what SM will accept. It was prepared from the real YACC grammar using the get_grammar utility.

```
    YACC grammar from ?
%{
%}

%start line

%token <charval>     WORD STRING
%token <floatval>    FLOAT
%token <intval>      INTEGER
%token
      '\n' '{' '}' '(' ')' '[' ']' CHDIR DEFINE DELETE DO EDIT ELSE
      FOREACH HELP HISTORY IF KEY LIST MACRO OVERLOAD PRINT PROMPT
      QUIT READ RESTORE SAVE SET SNARK
      STANDARD TERMTYPE VERBOSE VERSION WRITE
%token
      ANGLE ASPECT AXIS BOX CONNECT CONTOUR CTYPE CURSOR DATA DEVICE DOT
      DRAW ERASE ERRORBAR EXPAND FORMAT GRID HISTOGRAM
      IMAGE LABEL LEVELS LIMITS LINES LOCATION LTYPE LWEIGHT MINMAX
      NOTATION POINTS PTYPE PUTLABEL RANGE RELOCATE RETURN SHADE SORT SPLINE
      TICKSIZE USER WHATIS WINDOW XLABEL YLABEL
%token
      OLD ROW
%token
      '=' '+' '-' '*' '/' '<' '>' '!'  '&' '|' '?'  ':'
      ABS ACOS ASIN ATAN CONCAT COS DIMEN EXP FFT INT LG LN
      PI SIN SQRT SUM TAN
      POWER_SYMBOL

%left '|'
%left '&'
%left '=' '!'  '<' '>'
%left CONCAT
%left '+' '-'
%left '*' '/'
%left UNARY
%left POWER_SYMBOL

%type <charval>   number_or_word something string_or_space variable
      word_or_space
```

```
%type <floatval>  number optional s_expr
%type <intval>    fill integer_or_space plot sign
%type <pairval>   limit_pair
%type <t_list>    delim_list for_list list prompt comma_list
%type <vector>    expr pexpr

%%
line  :
      | line command
      ;

command :  '\n'
      | '!'
      | ANGLE expr
      | ASPECT s_expr
      | CHDIR WORD
      | CURSOR '\n'
      | CURSOR WORD WORD
      | IMAGE CURSOR
      | DATA WORD
      | EDIT {} WORD
      | DEFINE variable delim_list
      | DEFINE variable IMAGE
      | DEFINE variable READ INTEGER
      | DEFINE variable READ INTEGER INTEGER
      | DEFINE variable number_or_word
      | DEFINE variable ':'
      | DEFINE variable '?'  prompt
      | DEFINE variable '|'
      | DEFINE variable '(' expr ')'
      | DEFINE variable DELETE
      | DELETE
      | DELETE INTEGER
      | DELETE INTEGER INTEGER
      | DELETE HISTORY
      | DELETE HISTORY INTEGER
      | DELETE HISTORY INTEGER INTEGER
      | DELETE WORD
      | DEVICE INTEGER
      | DEVICE WORD
      | DO variable '=' s_expr ',' s_expr optional '{' list '}'
      | error
      | EXPAND expr
      | FFT number pexpr pexpr WORD WORD
      | FOREACH variable for_list '{' list '}'
      | FORMAT number_or_word number_or_word
      | FORMAT '\n'
      | {} graphic {}
```

```
| HELP
| HISTORY
| HISTORY '-'
| IF '(' s_expr ')' '{' list '}' '\n'
| IF '(' s_expr ')' '{' list '}' ELSE '{' list '}'
| IMAGE DELETE
| IMAGE WORD
| IMAGE WORD number number number number
| LEVELS expr
| LIMITS limit_pair limit_pair
| KEY
| LINES INTEGER INTEGER
| LIST DEFINE '\n'
| LIST DEFINE WORD WORD
| LIST DEVICE
| LIST EDIT
| LIST MACRO '\n'
| LIST MACRO WORD WORD
| LIST SET
| LOCATION INTEGER INTEGER INTEGER INTEGER
| LWEIGHT INTEGER
| MACRO WORD integer_or_space '{' list '}'
| MACRO WORD DELETE
| MACRO WORD INTEGER INTEGER
| MACRO EDIT WORD
| MACRO READ WORD
| MACRO DELETE WORD
| MACRO WRITE WORD '\n'
| MACRO WRITE WORD WORD
| MACRO WRITE WORD '+' WORD
| MINMAX WORD WORD
| NOTATION number number number number
| OVERLOAD variable INTEGER
| PRINT word_or_space string_or_space delim_list
| PROMPT
| PTYPE INTEGER INTEGER
| PTYPE WORD
| PTYPE '(' expr ')'
| PTYPE delim_list
| QUIT
| RANGE number number
| READ EDIT WORD
| READ OLD WORD WORD
| READ ROW WORD number_or_word
| READ WORD number_or_word
| READ delim_list
| RESTORE word_or_space
| RETURN
```

```
      | SAVE word_or_space
      | SET DIMEN '(' WORD ')' '=' number_or_word
      | SET HELP WORD
      | SET WORD '=' expr
      | SET WORD '=' expr '?'  expr ':'  expr
      | SET WORD '=' expr IF '(' expr ')'
      | SET WORD '=' s_expr ',' s_expr optional
      | SET WORD '=' WORD '(' comma_list ')'
      | SET WORD '[' s_expr ']' '=' expr
      | SNARK
      | SORT delim_list
      | SPLINE WORD WORD WORD WORD
      | TICKSIZE number number number number
      | TERMTYPE WORD integer_or_space
      | VERBOSE number
      | VERSION
      | WINDOW number number number number
      | WORD
      | WRITE HISTORY WORD
      | WRITE STANDARD
      | WRITE WORD
      ;

comma_list :  number_or_word
      | comma_list ',' number_or_word
      ;

expr  :  delim_list
      | expr CONCAT expr
      | WORD
      | STRING
      | WORD '[' s_expr ']'
      | '(' expr ')'
      | '-' expr %prec UNARY
      | '(' expr '?'  expr ':'  expr ')'
      | ABS '(' expr ')'
      | ACOS '(' expr ')'
      | ASIN '(' expr ')'
      | ATAN '(' expr ')'
      | COS '(' expr ')'
      | EXP '(' expr ')'
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | expr POWER_SYMBOL expr
      | expr '=' '=' expr
      | expr '!'  '=' expr
```

```
      | expr '&' '&' expr
      | expr '|' '|' expr
      | expr '>' expr
      | expr '>' '=' expr
      | expr '<' expr
      | expr '<' '=' expr
      | HISTOGRAM '(' expr ':'  expr ')'
      | IMAGE '(' expr ',' expr ')'
      | INT '(' expr ')'
      | LG '(' expr ')'
      | LN '(' expr ')'
      | number
      | PI
      | SIN '(' expr ')'
      | SQRT '(' expr ')'
      | TAN '(' expr ')'
      ;

delim_list :  '<' list '>'
      | '{' list '}'
      ;

fill  :  SHADE
      | SHADE HISTOGRAM
      ;

for_list :  '(' list ')'
      | '{' list '}'
      ;

graphic :  AXIS number number number number INTEGER INTEGER INTEGER
      | BOX
      | BOX INTEGER INTEGER
      | BOX INTEGER INTEGER INTEGER INTEGER
      | CONTOUR
      | CTYPE '=' expr
      | CTYPE INTEGER
      | CTYPE WORD
      | DOT
      | DRAW number number
      | DRAW '(' number number ')'
      | ERASE
      | ERRORBAR pexpr pexpr expr INTEGER
      | GRID integer_or_space integer_or_space
      | LABEL
      | LTYPE number
      | LTYPE ERASE
      | plot pexpr pexpr
```

```
        | plot pexpr pexpr IF '(' expr ')'
        | PUTLABEL INTEGER
        | RELOCATE number number
        | RELOCATE '(' number number ')'
        | fill INTEGER pexpr pexpr
        | USER INTEGER
        | XLABEL
        | YLABEL
        ;

integer_or_space :  INTEGER
        | '-' INTEGER
        |
        ;

limit_pair :  pexpr
        | number number
        ;

list :
        | list number_or_word
        | list STRING
        | list '\n'
        ;

number :  DIMEN '(' expr ')'
        | sign FLOAT
        | sign INTEGER
        | SUM '(' expr ')'
        | WHATIS '(' something ')'
        ;

number_or_word :  number
        | WORD
        ;

optional :
        | ',' s_expr
        ;

pexpr :  WORD
        | '(' expr ')'
        ;

plot  :  CONNECT
        | HISTOGRAM
        | POINTS
        ;
```

```
prompt :
      | delim_list
      ;

sign  :  '-'
      |
      ;

s_expr :  expr
      ;

something :
      ;

string_or_space :
      | STRING
      ;

variable :
      ;

word_or_space :
      | WORD
      ;

%%
```

Appendix V. Two-Dimensional Graphics

There is limited support for 2-dimensional graphics in SM, and we do not expect to make many extensions in this area. Currently it is possible to read an unformatted image, to draw contours, to extract values from the image into vectors, and to obtain image values with a cursor. We expect to support half-tone imaging in the nearish future. (Cynics will note that this phrase has appeared in previous releases of the manual).

The problem of specifying data formats for images is difficult, and we have adopted an approach of using a 'filecap' file to describe the unformatted files. This allows the user to write the file using C or Fortran (or, presumably, lisp) and then use SM to read the data. The name of the entry in the `filecap` file is given by the variable `file_type`, which may be given in your **.sm** file if you use the standard `startup` macro. The format of data on disk is first the x- and y- dimensions of the image, then the data in row-ascending order. The exact statements used to write the data may depend on the chosen value of `file_type` (or vice-versa). A description of the `filecap` fields comes at the end of this appendix; most users should never have to change this file.

The command used to read an image is IMAGE, and you may optionally specify the x and y range covered by the data (*i.e.* IMAGE **`datafile xmin xmax ymin ymax`**). Only the region of the image lying within the current limits is plotted when contouring - just like any other graphics operation in SM. Images may be deleted with the DELETE IMAGE command. To extract values from an image, use the special expression `IMAGE(`$\mathtt{vec}_x$`,`$\mathtt{vec}_y$`)` which has the value of the image at the points ($\mathtt{vec}_x$,$\mathtt{vec}_y$).

You can extract variables from the image's header using the command **`DEFINE name IMAGE`**. Currently this only works for `X0`, `Y0`, `X1`, and `Y1`, and for FITS headers (note that SM is case sensitive).

If you want to use non-interactive SM (*i.e.* write your own command interpreter), there is a call (**`defimage`**) to define your data array to the contouring package, but this is of course not available interactively.

Filecap.

Filecap is similar in graphcap or the Unix termcap, and in fact the 'filecap' entries may be physically in the same file as graphcap. Filecap can be specified as a list of files to be searched in order. The fields in filecap are used to specify the data type of the file, the record format of the file (record-length, inter-record gaps, etc.) and the header used (length of header, offset of desired items). Note that these are 'unformatted' files, as written by C write() or fortran write(num) statements.

The current filecap is as follows:

```
#
```

```
# Filecap describes unformatted file formats to SM. The syntax is
# identical to termcap or graphcap files.
#
c|C|C files:\
    :HS#8:nx#0:ny#4:
ch|CH|C files with headers:\
    :HS#24:nx#0:ny#4:x0#8:x1#12:y0#16:y1#20:
fits|cfits|FITS|CFITS|C FITS files:\
    :DA=fits:RL=2880:
no_header|Files with no header, will prompt for nx, ny:\
    :HS#0:
unix|UNIX|Fortran unformatted files under Unix on a Vax:\
    :HS#-1:nx#0:ny#4:RS#4:RL#-1:RE#4:
unix_int|UNIX_INT|Like unix, but integer*4:\
    :DA=int:tc=unix:
unix_short|UNIX_SHORT|Like unix, but integer*2:\
    :DA=short:tc=unix:
vms_var|VMS_VAR|Fortran unformatted under VMS, recl=512 (not 2048):\
    :HS#8:RS#2:RE@:RL#2048:nx#0:ny#4:
vms_fixed|VMS_FIXED|Fortran unformatted under VMS, recordtype=fixed:\
    :HS#-1:RS#0:RL#-1:RE@:nx#0:ny#4:
```

Comment lines start with a #, continuation lines start with white space (a tab or a space) and \ may be used to continue an entry on to the next line. The first few fields in an entry are separated by |, and are alternative names for the same entry. For example, fortran unformatted files under Unix may be referred to as `unix` or `UNIX`. Fields are separated by colons, and are of the form `:CC#nnn:` for numbers, and `:SS=str:` for strings. Omitted fields may be specified as `:CC@nnn:`, or simply omitted. Filecap capabilities currently used are:

DA	DAta type	Type of data in file (string)
FS	File Start	Unwanted bytes at start of file
HS	Header Size	Size of header
RE	Record End	Unwanted bytes at end of record
NS	No Swap	Don't swap bytes for FITS files
RL	Record Length	Number of useful bytes per record
RS	Record Start	Unwanted bytes at start of record
nx	Number X	Offset in file of X size of data
ny	Number Y	Offset in file of Y size of data
x0	X 0	Coordinate at start of X axis
x1	X 1	Coordinate at end of X axis
y0	Y 0	Coordinate at start of Y axis
y1	Y 1	Coordinate at end of Y axis

In addition, HH is supported as an archaic form of HS.

In terms of these quantities a file will look like this:

FS	HS	RS	RL	RE	RS	RL	RE	...

The distinction between single and double lines is purely visual. As mentioned below, HS can be negative and is then taken as the record length of the header RL_H, in which case the file will be like:

FS	RS	RL_H	RE	RS	RL	RE	RS	...

Most parameters are optional, and will default to 0. If RE or RS is specified, you must give RL as well. If you specify it as -ve, then we'll look for it from the operating system. You must provide a value for HS (or HH if you're old fashioned), if it is negative we'll assume that even the header has a record structure, with RS and RE just like any other record. It's record length will be taken to be RL, if RL is positive, otherwise we'll find it from the operating system. If HS and RL are both negative there is no reason why the record length of the header should be the same as that of the data. If HS is zero you will be prompted for the values of nx and ny, otherwise they must both be present and give the offset of the values of the x and y dimensions in the file, relative to the start of the header (if you set HS to be zero you can specify nx and ny on the command line, but they m,ust be on a separate line, either a real separate line, or following a \n). Note that HS excludes FS, so HS will usually be 2*sizeof(int) irrespective of the value of FS, and nx will usually be 0. If HS is negative, then the nx and ny offsets also ignore RS, *i.e.* nx is still usually 0. Possibilities for `DA` are `char`, `float` (default), `int`, `long`, and `short`, all as in C, and also `fits` for FITS format data. (If you don't know what FITS format is, don't worry about it. It's a style of header and record structure used for data transport in astronomy. If you do know about FITS, then we assume that each record is 2880by long, although possibly with RS and RE non-zero. The header is processed for the size of the file and the value of BITPIX. If the file is not SIMPLE, it is taken to be 4by floating point data. BZERO and BSCALE are interpreted correctly. X0, x1, y0, and y1 are specified as the keywords X0, X1, Y0, and Y1 respectively. FITS files on a machine with vax byte order are supposed to be byte swapped, but you can override this by specifying the `NS` capability which stops SM from doing any swapping). If `x0` and `x1`, or `y0` and `y1` are omitted, the range of values on the appropriate axis is taken to be 0 to nx-1 (or ny-1). You can override these values with the `IMAGE` command.

For example, suppose I wrote a file using the Unix f77 compiler, with some code that looked like:

```
      integer *4 nx,ny,arr(10,20)
c
      nx = 10
      ny = 20
      write(8) nx,ny
      write(8) arr
```

(Omitting opening unit 8 as an unformatted file, and filling the array with data). Then I could read it in SM by setting `file_type` to be unix_int. The filecap entry indicates that the length of the header is to be obtained from unix (in fact from the file, it'll be the record length of the header), as is the record length. Both the start-of-record (RS) and end-of-record (RE) gaps are 4by long (they in fact contain the record length, but you needn't know that). The number of x-records is

at zero offset in the file, and y-records is at offset 4. In other words, allowing for FS being 0 and RS being 4, nx occupies bytes 4-7 in the file, and ny 8-11. The data is taken to be 4-byte integers (integer*4 to fortran). In fact, the first record consists of only the values of nx and ny, so the record length of the first record is 8by, and part of an equivalent filecap entry would be

```
:FS#4:HS#12:
```

where I have interpreted RS as FS, and added the RE onto the end of the header length HS. If I had written the data out line-by-line in a do loop, the file type would still be unix_int, but the record length actually used by SM in reading the file would be different.

As another example, I use an image processing system called Wolf that used to use files with the arcane structure of 200bytes of header, then the x- and y-size of the file, given as ints, then more header up to a total offset of 1024 bytes from the start of the file, then the data written as short integers. The corresponding filecap entry is

```
wolf|Wolf-IfA files:\
   :HS#1024:nx#200:ny#204:DA=short:
```

which is pretty simple really.

Appendix VI. Termcap

Termcap is a Unix idea, that makes it possible to write software that runs on a wide range of different terminals. The implementation that SM uses contains no Unix source code. The idea is similar to graphcap or filecap – each property of the terminal is given as a field in a file. For example, the string to clear to end of line is called `ce`, and for ansi terminals is given as `ce=^[[K`. The format for termcap files is described under 'graphcap', which is identical. Our termcap is identical to the Unix one, so for details see section 5 of the Unix manual. The file to use as a termcap file is specified as `termcap` in your **.sm** file, or as the environment variable `TERMCAP` (logical variable to VMS). If SM can't open `TERMCAP` as a file, it is taken to be the value of the entry for your terminal. If it doesn't begin with a colon, it is treated like a termcap file, and searched for the desired terminal. If it does begin with a colon, it is simply taken to be the termcap entry to use. A `TERMCAP` variable takes precedence over an entry in your **.sm** file. As for graphcap, a list of termcap files can be specified, to be searched in order.

A full termcap entry can be pretty long, but fortunately SM only uses a small subset. (The rest are needed by full scale screen editors, but we haven't written one.) In particular we use:

ce	Clear End	Delete to end of line
ch	Cursor Horizontal	Move cursor within line
cm	Cursor Movement	Move cursor around screen
cn	ColumNs	Number of columns on screen
dc	Delete Character	Delete character under cursor
is	InitialiSe	Initialise terminal
ks	Keypad Start	Initialise Keypad
ks	Keypad End	Undo Keypad Initialisation
kd	Kursor Down	Move cursor down one character
kl	Kursor Left	Move cursor left one character
kr	Kursor Right	Move cursor right one character
ku	Kursor Up	Move cursor up one character
k1	Key 1	Key sequence emitted by PF1 key (also `k2`, `k3`, and `k4`)
li	Lines	Number of lines on screen
pc	Pad Character	Pad character, if not NIL

All of these are strings except `co` and `li` which expect numbers, and `pc` which expects a single character. The editor uses all of these except `co`, although the `k` ones are only provided as a convenience to you, mapping the arrow keys. In fact, these arrow keys may supercede desired bindings such as ^K to delete to end of line (*e.g.* on a televideo 912). If this happens, use the `EDIT` or `READ EDIT` commands to fix things up.

The cursor addressing strings `ch` and `cm` use a variant of the C printf %. Consider the string as a function, with a stack on which are pushed first the desired column, and then the desired line (so the line is on top). The possible % formats operate on the stack, and pop it after output.

```
%d   As in printf, zero origin
```

```
%2    Like %2d
%3    Like %3d
%.    Like %c
%+x   Add x to value, then '%.'
%>xy  If value > x add y (no output)
%r    Interchange line and column (no output)
%i    Increment line and column by 1 (no output)
%%    Output a single%
```

We do not support %n, %B, or %D out of pure laziness. A couple of examples might help. To go to (5,60) a vt100 expects to receive the string ^[[5;60H, so the termcap entry reads :cm=\E[%i%d;%dH:. We need the %i to go to one-indexed coordinates, and the escape is written \E. The :'s delimit the termcap entry. A Televideo 912 expects to be given first ESC=, then the coordinates as characters, where ' ' is 0, '!' is 1, and so forth through the ascii character set. To convert a given number to this form we add ' ', so the termcap entry is :cm=\E=%+ %+ :.

If an entry begins with a number, it gives the number of milliseconds of padding that the terminal requires (it is optionally followed by an *, which we can and do ignore). If the terminal requires a pad character that is not simply ascii NIL it should be given as pc, so to use 'a' as a pad character specify :pc#97:. The baudrate is taken to be 9600, unless you specify it as the ttybaud variable in your **.sm** file. We have never yet seen SM have any trouble with padding, so we wouldn't worry about any of this if we were you.

You might wonder why we need both ch and cm. The simple answer is that we don't, but that we do want one of them. Because cm moves the cursor to a specific line, its use requires that SM remember which line you are on which can be tough what with switching to and from graphics mode. We therefore assume that you are on the last line of the terminal, as set by li or explicitly by the TERMTYPE command. If you provided ch this problem wouldn't arise but many terminals don't support such a capability and we have to code around this deficiency. To summarise: use ch if you can, failing that use cm.

There is a problem with using the last line of the screen as the SM command line, and this is that some terminals use the same screen for graphics as for text, and your graph will merrily scroll away as you type. The graphcap GD (Graphics Disable) command can be set to take you to the top of the screen, but cm won't keep you there. Your only chance is to disable cursor motion, either by setting ch to 'disabled', or by specifying a negative screen size to TERMTYPE which has the same effect. You can still use the editor, but it'll be slower. If you still have trouble, try TERMTYPE dumb, but dumb really is a bit brain damaged.

Appendix VII. Adding New Devices, and Porting to New Machines

Adding New Devices

As we discussed extensively in Appendix II, most plotting devices can be accommodated within the framework of stdgraph/graphcap without writing a line of C; raster devices can use the raster device driver. Sometimes, however, this is not possible and you must write a real driver. An example would be making SM run native under X11; running under a terminal emulator can be done with graphcap.

If you do make any changes, write any new drivers, or modify graphcap (except trivially) *please* send us copies of your modifications.

Let us assume that you really do need a new driver, how do you go about it? SM communicates with devices solely through some external variables (defined in sm.h) and a structure called `DEVICES` which is defined in `devices.h` and declared in `plotsub/devices.c`. The current definition looks like this:

```
typedef struct {                    /* output device dependent functions */
        char    d_name[40];         /* name of device */
        int     (*dev_setup)();     /* initialisation */
        void    (*dev_enable)(),    /* enable graphics */
        void    (*dev_line)(),      /* draw a line from x1, y1 to x2, y2 */
        void    (*dev_reloc)(),     /* relocate current position */
        void    (*dev_draw)(),      /* draw a line from current pos to x,y */
        int     (*dev_char)(),      /* hardware characters */
        int     (*dev_ltype)(),     /* hardware line type */
        int     (*dev_lweight)(),   /* hardware line weight */
        void    (*dev_erase)(),     /* erase graphics screen */
        void    (*dev_idle)(),      /* switch from graph to alpha mode */
        int     (*dev_cursor)(),    /* cursor display */
        void    (*dev_close)(),     /* close device */
        int     (*dev_dot)(),       /* draw a single pixel dot */
        int     (*dev_fill_pt)(),   /* draw a filled point */
        void    (*dev_ctype)(),     /* set colour of line */
        int     (*dev_set_ctype)(), /* set possible colours of lines */
        void    (*dev_gflush)();    /* flush graphics */
} DEVICES;
```

Note that this is the DEVICES structure at the time of writing; be sure to see that it is still correct when you write your driver! To add a new device to SM, all you have to do is to write functions to fill as many of the slots in `DEVICES` as possible, then add an element to the array devices[] in devices.c. After recompiling, SM will recognise your device by the name `d_name` that you gave it in devices[]. Traditionally the functions have the same name as those given above with the `dev` replaced by some mnemonic for your new device. You should surround both your driver and the entry in devices[] with #ifdef's so that all I have to do to totally lose your device is to not #define it to the compiler. In writing your device you may want to

use graphcap to give it various snippets of information. This can be done; see the imagen driver for an example.

You should include the file **sm_declare.h** (which automatically includes **options.h**) which provides declarations for all SM functions (including prototypes if the compiler supports them), and you should add tour own declarations to this file.

If you don't provide some of the functions, SM will try to emulate them in its software. For example, of your ltype function returns -1, then we will chop your lines for you. There are a set of functions for `nodevice` that have the correct types, you can use them for functions that you don't want to provide (*e.g.* if you don't support `dev_ltype` you can use `no_ltype`). Some of the routines, *e.g.* dev_char are sometimes passed NULL arguments to inquire if they can provide particular capabilities. In the following paragraphs we discuss all the functions, their arguments, what they do, and what they return.

All coordinates are given in screen units, where the device is 32768 pixels along a side.

dev_setup(str)

(char *str;) Open the plotting device. Str contains the rest of the command line, so if the `DEVICE` command was `DEVICE newdev abcdefg zyxwvut`, setup will be passed `abcdefg zyxwvut`. If the device can't be opened for some reason setup should return -1, otherwise 0. Setup has a number of book-keeping responsibilities: setting the value of termout to 1 if the device is a terminal, otherwise 0; setting the value of ldef to the maximum spacing between lines if they are to appear as one thick line rather than a set of parallel ones, (used if we have to emulate `LWEIGHT`); setting the variables xscreen_to_pix and yscreen_to_pix to give the conversion from SCREEN to device coordinates; calling default_ctype with the name of the default colour for lines (*e.g.* `default_ctype("white")`). There is no need to look in the **.sm** file for a `foreground` entry, this is done for you (by set_dev()) after dev_setup returns, but you should deal with any `background` entry if you feel so inclined. There is a function parse_color that converts a colour name into r, g, and b values (see the sunwindows setup function for an example of its use) that may help.

dev_enable()

Enable plotting on the device. For a graphics terminal this means switching from alpha to graphics mode, but may well not be required for other devices. No arguments, no return value.

dev_line(x1,y1,x2,y2)	(int x1, y1, x2, y2;) Draw a line from (x1,y1) to (x2,y2). No return value.
dev_reloc(x,y)	(int x,y;) Move the plot marker to (x,y). No return value.
dev_draw(x,y)	(int x,y;) Draw a line from the current plot marker to (x,y) which becomes the new plot marker. No return value.
dev_char(str,x,y)	(char *str; int x,y;) Write the string at the position (x,y). The position given is just to the left of the first character, at about the height of the centre of a capital letter. Return 0 if you support hardware characters, otherwise -1 in which case we'll use the software character set instead. If str is NULL, this is just an enquiry as to whether the device supports a hardware character set. Char is also called (with NULL) whenever the value of expand or angle is changed, to allow you to adjust the sizes of cheight and cwidth (the height and width of a character in SCREEN units) for a hardware character set, as the sizes of hardware sets can be very different from the SM fonts. You should only change cheight and cwidth if we are really going to use your hardware characters (expand == 1, angle == 0).
dev_ltype(i)	(int i;) Set the `LTYPE` to be i, as in the interactive command. Return 0 if you support the ltype asked for, otherwise -1. If you can't support a linetype, make sure that you are drawing solid lines, so that SM can emulate it for you. `LTYPE` 10 is special, it is generated by the `LTYPE ERASE` command, and asks you to delete lines as they are drawn, `LTYPE 11` is used to indicate the end of `LTYPE ERASE` mode.
dev_lweight(i)	(int i;) Set the `LWEIGHT` to be i, as in the interactive command. Lines with `lweight` 0 should be the natural thickness of a line on your device, higher `lweight` lines are usually drawn with a thickness of `LDEF*lweight` in SCREEN coordinates. Return 0 if you support the lweight asked for, otherwise -1.
dev_erase()	Erase the screen. No return value.
dev_idle()	Return to alpha mode, the opposite of dev_enable(). No return value.

dev_cursor(x,y)	(int *x,*y) Find the current position of the cursor, if there is one. If there is, the return value is the character struck by the user, otherwise return -1. If your device has a mouse, you should try to make the buttons correspond to 'e', 'p', and 'q' from left to right. If you only have one button, 'p' is probably the best choice.
dev_close()	Close the device, the opposite of dev_open(). No return value.
dev_dot(x,y)	(int x,y) Draw a dot at (x,y), (*i.e.* a real dot, not like the `DOT` command). You'll have to do the move to (x,y) yourself. Return 0 if you can draw dots, otherwise -1.
dev_fill_pt(n)	(int n;) Draw a filled point at the current position. This is the routine that draws `PTYPE n 3` points. Remember to allow for `EXPAND` and `ANGLE`. Return 0 if you can draw the point, otherwise -1;
dev_ctype(r,g,b)	(int r,g,b;) Set the current line colour. The interpretation of (r,g,b) depends on whether you have a dev_set_ctype function. If you don't, then r, g, and b are the red, green, and blue intensities in the range 0-255. If you do, then r is an index into the array defined by set_ctype and g and b are irrelevant. No return value.
dev_set_ctype(col,n)	(COLOR *col; int n;) Define a set of colours to be accessed by set_ctype. Col is an array of n elements, each of which consists of 3 unsigned chars (COLOR is defined in mongo.h) corresponding to red, green, and blue in the range 0-255. Ctype will be used to access this array to change colours. Return 0 if you do support set_ctype, otherwise -1. If col is NULL this is just an enquiry. If you are asked for more colours than you are prepared to support, you should scale the request in a user transparent way (*e.g.* if she wants 256 colours, and you are only giving her 128, you should arrange that dev_ctype divides her requests by 2 so it looks as if she got all 256).
dev_gflush()	Flush graphics, update graphics on screen. No return value.

If you are still confused, look at some of the hardware drivers that are already available.

Porting to New Machines

Porting to new Unix machines should be relatively simple. If the machine runs BSD Unix the only changes that should be necessary are to the exception handler routines in main.c, to reflect the error conditions signalled by your new machine. For a Sys V Unix, you'll have to edit **options.h** to define SYS_V. Otherwise, your only problems should be hidden bugs which flourish in new architectures.

Otherwise there isn't all that much that I can tell you. SM runs on Unix (BSD and SysV) and VMS machines, and the places where the code is #ifdef'd for those operating systems is your best bet for places where there are machine dependencies. That said, there are a few obvious problem areas.

First consider getlchar, which is the routine that reads terminal input. It has to be able to get one character a time from the terminal, and not interpret control characters. This is obviously operating system dependent (CBREAK under 4.3BSD, PASSALL under VMS, ...). Getlchar expects to be passed ^A to start operations, and EOF to end them. Anything else means 'return a character please'. The terminal output functions used by stdgraph are also machine dependent, for example they use QIO calls under VMS.

As mentioned above, the exception handlers try to interpret the exceptions that they receive and this may require a little modification in main.c.

Hardcopy devices need the system() system call to send commands to the operating system, and if you don't have one you are in trouble.

In a few places SM needs to make assumptions about how file names are put together, and these will need changing. On a similar note, some operating systems (*e.g.* VMS) are very picky about opening files and you may have to be careful. Note the use of the DT graphcap capability to signal particular requirements to the programme.

A machine that didn't use ascii would be rather a nuisance as makeyyl assumes that the characters for A-Z are contiguous, and although this could be easily fixed I am not sure that other problems wouldn't arise.

A final rather horrifying thought is that Bison might not compile on a machine that doesn't have YACC as an alternative. I don't know how to fix that, you'd just have to hope for the best, or else run Bison/YACC on a different machine and copy over control.c.

Appendix VIII. Macro Libraries

This appendix describes the collection of useful macros, written by a variety of people, that are to be found in the default macro directory, *i.e.* the directory pointed to by the `macro` entry in your **.sm** file. If ever you write a useful (or simply clever) macro, why not send it to us for inclusion in the next version of SM?

The macros are arranged in a number of files which may be read using the `load` macro. For example, to load the file `fonts` type `load fonts`. To forget a set of definitions, use `unload`. Under Unix, there is a macro `lsm` that can be used to list the contents of macro libraries, *e.g.* `lsm utils`. It is more-or-less complete depending on the value of `VERBOSE`, just like `LIST MACRO`.

The list that follows gives the name and a one-line synopsis of all the macros in the default files as of the date of this manual. The full text of any macro may be examined via the `help <macroname>` command in SM; the default files may be printed for those who desire a hardcopy.

```
                        file cover in directory macro
cover                    # draw the cover
                       file default in directory macro
batch                    ## run the history buffer, but don't delete from history list
bell                     ## ring terminal bell
calc                     ## evaluate an expression
cd                       ## change directories
compatible               ## define macros to be compatible with Mongo
declare                  ## declare a vector $1:  declare name size
del                      ## delete last command from history list
del1                     ## don't put command on history list
echo                     ## write to terminal
ed                       ## edit a macro, or the previous one if no argument
edall                    ## edit history buffer
edit_all                 ## edit history buffer
edit_hist                ## edit the history list
gripe                    ## complain to Robert
h                        ## get help
head                     ## print the top of the current data (or other) file
hm                       ## help with the last macro edited
hv                       ## help with variable
insert                   ## insert text after line $1
load                     ## load macros in default directory
load2                    ## load macros in (second) default directory
ls                       ## list macros
lsm                      ## list macros in a file in default macro directory
lsv                      ## list variables
q                        ## check, then quit
re                       ## macro read
reset_ctype              # Reset the default ctypes (except "default")
sav                      ## save to a file $1, don't save from files '$mfiles'
```

```
show                    ## show current values of various things
startup                 ## macro invoked upon startup
undef                   ## undefine a variable
undo                    ## undo [macro] :  undo (i.e.  erase lines drawn by) macro $1
unload                  ## forget macros from a file
v                       ## set verbosity
wr                      ## macro write
                      file demos in directory macro
square                  # sum the Fourier series for a square wave, using $1 terms
colours                 # draw a circle in a number of colours until Ĉ stops it
crings                  # draw a set of coloured circles
scribble                # use cursor to draw a line, and then shade the interior
shading                 # draw an ammonite
sundial                 # draw a sundial, allowing for the analemma
                      file fonts in directory macro
fonts                   # draw the font table
TeX_defs                # draw the ''TeX'' definitions
make_char               # help create a new character
                     file fourier in directory macro
#
#                       Macros to make dealing with complex numbers for FFT's easier
#                       Assumes that complex vector 'name' is represented by two vectors
#                       called name_r and name_i
#
fft                     # Direct FFT: fft name name
ifft                    # Inverse FFT: ifft name name
cadd                    # Add complex numbers:  $1 = $2 + $3
cdiv                    # Divide complex numbers:  $1 = $2/$3
cmod                    # Modulus:  $1 = |$2|
cmult                   # Multiply complex numbers:  $1 = $2*$3
csub                    # Subtract complex numbers:  $1 = $2 - $3
imag                    # Imaginary part:  $1 = Im($2)
real                    # Real part:  $1 = Re($2)
                    file mongo in directory macro
```

(omitting those that are simply abbreviations, like ang)

```
da                      ## set data file
dev                     ## set device
dra                     ## draw, accepting expressions
ecolumn                 ## define vector error_col as error vector
end                     ## quit, not on history
era                     ## erase screen, not on history
lis                     ## list history, not on history
hard                    ## make a hardcopy of what you type (or get by history)
hardcopy                ## close the old device and set dev type to 0
hcopy                   ## hcopy [printer] [l1] [l2]:  Make hardcopy of playback buffer
hmacro                  ## hmacro [macro] [printer] :  make hardcopy of 'macro' on 'printer'
identification          ## write an id to the top right hand corner of screen
```

```
input          ## execute an Mongo file
pcolumn        ## set point column (Mongo)
playback       ## define "all" from buffer, and run it
read_all       ## read a macro file, putting 'all' onto the history buffer
read_hist      ## read history from a file
read_old       ## read an Mongo file onto the history buffer
rel            ## relocate, accepting expressions
save_all       ## write the playback list to a file (use sav instead)
terminal       ## device
toplabel       ## put label at top of plot
xcolumn        ## read a column into vector x (Mongo)
xlogarithm     ## take log of vector x (Mongo)
ycolumn        ## read a column into vector y (Mongo)
ylogarithm     ## take log of vector y
```

file **stats** in directory **macro**

```
cgauss         # evaluate a Cumulated Gaussian :  N($mean,$sig)
draw_KS        # Draw a cumulated curve, for looking at KS statistics
erfc           # calculate complentary error function erfc($1)
factorial      # Use Stirling's formula to calculate a factorial ($1)!
gauss          # evaluate a Gaussian :  N($mean,$sig)
lsq            # do a least squares fit to a set of vectors
lsq2           # do a least squares fit to a set of vectors, errors in x and y
prob_KS        # probability of getting a given value of the KS statistic
prob_wilc      # return probability in $$2 that x exceeds $1 from Wilcoxon
rxy            # find Correlation Coefficient for two vectors
smirnov1       # calculate 1 sided Kolmogorov-Smirnov statistic for vector
smirnov2       # calculate 2 sided Kolmogorov-Smirnov statistic for vectors
spear          # calculate Spearman rank correlation coefficient for 2 vectors
stats          # stats vector mean sigma kurtosis :  calculate $mean $sigma etc
stats2         # stats vector weights mean sigma kurtosis
stats_med      # stats_med vector median SIQR : calc $median $SIQR from vector
wilcoxon       # calculate Wilcoxon statistic for 2 vectors
```

file **utils** in directory **macro**

```
alpha_poi      # alpha_poi x y z.  Like poi x y, but use z as labels for points
arrow          # use the cursor to define an arrow.
barhist        # draw a bar histogram
boxit          # use the cursor to define a box, and draw it
circle         # draw a circle, centre ($1,$2) radius $3
cumulate       # find the cumulative distribution of $1 in $2
draw_arrow     # draw an arrow from ($1,$2) to ($3,$4)
draw_box       # draw a box, defined by two corners
error_x        # draw x-error bars:  error x y size
error_y        # draw y-error bars:  error x y size
get_hist       # get_hist input output-x output-y base top width
gauss          # evaluate a Gaussian :  N($mean,$sig)
get            # syntax:  get i j.  Read a column from a file.
interp         # Linearily interpolate $3 into ($1,$2), giving $4
interp2        # Linearily interpolate $3 into ($1,$2), giving $4
```

```
is_set          # define variable $$1 if the $3'rd bit is set in $2
logerr          # syntax:  logerr x y error, where y is logged, and error isn't
mconcat         # Concatenate 2 macros, optionally renaming result
modulo          # find $1 modulo $2
pairs           # pairs x1 y1 x2 y2.  connect (x1,y1) to (x2,y2)
polar           # draw a circle as an 'axis' for polar coordinates
reverse         # reverse the order of a vector
save_vec        # put the definition of a vector onto the history list
shade_box       # shade a box, spacing $1, defined by two corners
shed            # shade region between x y and x2 y2 with n lines
simp            # Simpson's rule integration:  simp answer x y
smooth          # boxcar smooth a vector
smooth2         # smooth a vector with a given filter
upper           # define a variable giving an 'upper limit' symbol
vfield          # plot a vector field:  vfield x y len angle
```

Appendix IX: Tips for Mongo Users

Differences from Mongo

SM differs in a number of ways from Mongo, and these fall into three groups: those which are enhancements, those which are generalisations, and those which are simply incompatibilities. We do not feel that there is a fourth group for degradations. For those users of Mongo intimidated by change, we note that in most cases it is possible to ignore the enhancements by using a macro presented in the next section. This macro redefines commands to reproduce the old syntax; for example `limits` is defined to mean `LIMITS x y`. It is also possible to read Mongo files using the `READ OLD` command, and the macros `input` and `read_old` based upon it. The following list of enhancements in not complete; See the distribution notes from the current release of SM.

Enhancements:
Any number of vectors may be defined.
Vectors may be manipulated arithmetically.
Vectors are named.
Vectors may be defined from the keyboard, using `DO` loops or expressions.
Vectors may be defined using the cursor.
Any vector may be used for plotting.
Any vector may be used for the `PTYPE` or `ERRORBAR` commands.
A history feature is implemented.
The playback buffer may be edited.
Macros may be defined from the keyboard, and edited.
A `DO` construct is available.
A `FOREACH` construct is available.
Character strings may be read from a file, and used freely as labels or names.
Data may be read from rows as well as columns in files.
Only those parts of a vector satisfying a logical condition need be plotted.
Vectors may be sorted or fit with splines.
Macros exist for doing least square fit to sets of points, constructing cumulative distributions and histograms, drawing circles, and shading regions.
All devices have the same range of device coordinates, 0-32767.
The entire SM environment may be saved for later resumption
with `SAVE` and `RESTORE`.
The special variable `$date` expands to the current date and time.
You can define private point types.

There are also a few incompatibilities:
`DEFINE` is used to define *variables*; *macros* are defined using `MACRO`.
Macro arguments must be declared, and are referred to as `$n`, not `&n`.
The form `LIMITS` is not supported (it's meaningless); use
`LIMITS x y`, or the macro `lim`, mentioned above.
(But note that `READ OLD` allows for these, and makes suitable changes.)
`WINDOW` now takes 4 arguments.

The READ OLD command

`READ OLD` reads a Mongo file, converts its contents to a form acceptable to SM, and defines them as a macro. Any macro definition (*i.e.* from a line beginning `def` to a line beginning `end`) is converted to the SM form (*i.e.* **$s** not **&s**) and defined. The commands CONNECT, HISTOGRAM, LIMITS, and POINTS are converted to LIMITS x y, and so forth. ERRORBAR, ECOLUMN, and WINDOWS are also converted. READ OLD will fail if the Mongo file contains abbreviations such as xc for XCOLUMN, then your only hope is to define the same abbreviations. In many cases this will have already been done, for instance `xc` expands to `read x` . Comments (beginning !) are optionally converted to standard SM # comments (depending on how the file **read_old.c** was compiled.)

Note that it is advisable to convert these old Mongo macro files to SM macros, to enable you to take advantage of SM's features. You can do this by simply using READ OLD to read them into SM, and then MACRO WRITE or SAVE to write the converted macro out to disk.

There is also a macro equivalent of the old `INPUT` command.

input

```
1    # execute a Mongo file
     read old temp $1
     temp macro temp { delete }
```

The compatibility macro

This version of compatibility is more complete than in pre-version 2 SM, it also conflicts more strongly with normal SM operations.

The macro `compatibility` defines *mimics* for the Mongo commands which assume that the only vectors are `x` and `y`. We strongly recommend that you do *not* use this macro! If you want to use it anyway, commands like `limits alpha beta` will give syntax errors. You can turn compatibility mode off again with `compatibility 0`. The macro itself is a little complicated, it turns off the special meaning of (*e.g.*) `limits`, and replaces it with a macro that reproduces the old behaviour, in this case `LIMITS x y`. The new definitions are in the file **compatible** in the default macro directory, as specified in your **.sm** file. At the time of writing, the commands `connect`, `errorbar`, `histogram`, `limits`, `list`, `points`, `read`, and `window` are redefined to reproduce the old syntax. In addition, `help` is defined to not appear on your history buffer, and `define` is defined to create macros interactively. You might also be interested in other redefinitions of commands (*e.g.* `list` to mean list the playback buffer), if so look at 'overloading' in the index. It should be clear that this set of definitions could thoroughly confuse SM if you try to take advantage of its features; in the realm of compatibility mode, it is strictly *caveat emptor.*

compatible

```
11 ## define macros to be compatible with Mongo
   # If the argument is non-zero or omitted,
   # compatibility mode is turned on.
```

```
# note that some of these make it hard to use regular SM!
if($?1 == 0) {
    compatible 1
    RETURN
}
if($1 == 0) {
    MACRO DELETE "$!macro"compatible
}
FOREACH w { connect define errorbar help histogram limits \
      list points read window write } {
    OVERLOAD $w $1
}
IF($1) {
    MACRO READ "$!macro"compatible
}
#             So newline will end IF statement
```

Appendix X. The Available Fonts

These are a sub-set of the well-known Hershey fonts † and the available characters are listed in the following table, which were generated from within SM by saying `load fonts fonts`.

There are two separate ways to specify special characters to SM, by using a syntax very similar to TEX (the type-setting system created by Donald Knuth that we used for this manual), or the traditional Mongo way. You might ask what are the advantages of TEX? One is that sub- and super- scripts are handled much more naturally, making it much harder to type $M_{V=-8}$ when you meant $M_V = -8$. Another is that you no longer have to remember that θ is hidden in the Greek font as 'q', you can simply type `\theta`. A third would be that you may well know TEX already. Finally, enhancements to SM's strings (*e.g.* the ability to insert points in labels using `\point`, or the `\bf` font) are only supported in TEX mode.

To make SM understand TEX strings you must define the variable `TeX_strings` (if you put a line `TeX_strings 1` in your **.sm** file this will be done automatically). You can, of course, undefine it at any time to revert to the old-fashioned strings described above.

In the following description I assume that you don't already know TEX, TEXsperts will find a brief summary of the (minor) differences at the end of the description.

Let's start with an example:

```
label \pi^{\-21/2} = {\3\int}e^{-x^2}\,dx
```

will print a well-known result (You'll have to `RELOCATE` somewhere where the label will be visible first, of course). (If you want to try it now, you should be careful typing those ^, as they are special to the history editor, dealing with this is discussed below.) In this example the characters \, {, }, and ^ are special (and so is _ which wasn't used). Postponing \ for the moment, ^ means 'make the next group a superscript', _ means 'make the next group a subscript', where a group is either a single character, a single control sequence (wait a moment!), or a string enclosed in braces. So `A_a^{SM}B` would appear as $A_a{}^{SM}B$. A \ can serve one of two functions, either turning off the special meaning of the next character (so _ is simply a _ with no special significance), or to introduce a named 'control sequence'. These fall into two groups, those that change fonts and those that serve as abbreviations for single characters (*e.g.* `\pi` in the example). The font changes persist until the end of the string, or the current group, whichever comes first.

---

† created by Dr. A. V. Hershey at the U. S. National Bureau of Standards and illustrated in National Bureau of Standards publication NBS SP-24. The format used in the hershey_oc.dat file was originally due to James Hurt at Cognition, Inc., 900 Technology Park Drive, Billarica, MA 01821. It may be converted to any format *except* the format distributed by the U. S. NTIS (whatever it may be). We have to tell you all this for copyright reasons, but as distributed the fonts are in the public domain.

The available fonts are 'greek', 'old english', 'private', 'roman', 'script', and 'tiny'. They may be referred to either by a two-character control sequences (`\gr`, `\oe`, `\pr`, `\rm`, `\sc`, or `\ti`) or simply by the first character (*e.g.* `\r` for the roman font). In addition `\i` or `\it` can be used to make the current font italic (italics are turned off again either by a second `\it` or by grouping the first `\it` within {}). The 'bold' font `\b` or `\bf` is similar, in that it makes the current font bold, and can be toggled off with a second `\bf` if you didn't simply group it. I'd strongly recommend treating `\bf` and `\it` like any other font change, and group them rather than relying on this toggling action.

You can alter the size of the letters by using an escape such as `\6` which scales the current group (any font change is local to a group). `\6` corresponds to multiplying the size by 1.2^6 or about 3, `\-4` scales by $1/1.2^4$ or 0.48. This is similar to the 'magstep' used in scaling fonts in TEX. These scale factors are in addition to the expansion produced by going up or down (^ or _), or setting `EXPAND`.

Other control sequences either consist of one non-alphabetic character, or else a name consisting only of letters, so `\,` or `\palmtree` is valid but `\one2three` is not. If a alphabetic name is followed by a space, the space is treated as simply delimiting the name and is discarded. For example, `AB^{\alpha_\beta CD}` will appear as $AB^{\alpha\beta}CD$ (note that the space after `beta` disappeared). How do you make just a few characters italic (script, old english, etc.)? Try `ABC{\it DEF}GHI`. You can't read a subscript, and want it in 'tiny' font? Try `\Lambda_{\ti ab}`. All of the Greek letters are defined, as `\alpha-\omega,\Alpha-\Omega`, there are various mathematical symbols (*e.g.* `\int`, `\infty`, or `\sqrt`), some astronomical (*e.g.* `\AA` for Å), and some miscellaneous characters (*e.g.* `\snow` to draw a snowflake). A complete list of definitions follows the font table (it was generated by saying `load fonts TeX_defs`).

You can also define your own TEX definitions by using the special command `\def\name{value}` inside a label. It produces no output, but defines `name` to expand to `value`. For example, I could define `\TeX` to produce TEX by saying `LABEL \def\TeX{T\raise-200\kern-20E\raise200X}`. Once a definition has been made it is remembered forever (well, until you leave SM actually) whatever devices you plot on. You must make sure that all curly brackets are properly paired inside your definition. And no, you can't provide arguments (unlike real TEX).

You can insert points (such as would be drawn with `DOT`) into labels. The string `\point43` (or `\point 4 3`) will draw a point at the current position in the string, of `ptype` '4 3'. This sequence, from the `\` to the `3`, is treated as a single character as regards things like subscripts. If you want to specify an angle, use something like `\apoint 45 4 0`. You can also insert lines into labels with `\line`, *e.g.* `\line 1 1000` will draw a line of length 1000 (in screen units, so the screen is 32768 across), of ltype 1. You can change your default font by defining the variable `default_font`, either interactively (`DEFINE default_font oe`) or by putting a line in your `.sm` file: `default_font oe`. This affects regular as well as axis labels. It is also possible to explicitly move characters around with the commands `\kern #` (which moves the current plot pointer by # horizontally) and `\raise #` (which moves it vertically). The

distance # is specified in screen units (the whole screen is 32768 across) multiplied by the expansion currently in effect, and may be postive or negative.

Now for a few caveats: Firstly, because \n is a newline, you must type `\\nu` to get a ν. Secondly, the superscript character ^ is special to the history editor, so to type it interactively you must quote it with the **quote_next** key (usually ^Q or ESC-q, *i.e.* type `^Q^`). Alternatively, you could change your history character to some under-used character such as % or ' (which is the solution that I use: you can choose a new character such as ' by simply putting a line `history_char '` in your **.sm** file). Thirdly, TEX (and our pseudoTEX) are rather verbose and labels may not fit on one line. The solution is to continue the line by ending it with a \. This is probably best done within a macro, as the continuation line won't appear on your history list if typed at the prompt. You can currently have about 25 continuation lines (2000 characters). A final point will only worry TEXies, namely that the emulation isn't perfect: for example `\sum_i` won't put the `i` beneath the summation symbol.

For some devices with hardware fonts (for example, a Tektronix 4010), if **expand** is exactly 1, and **angle** is exactly 0, the hard fonts will be used for speed. To defeat this, try `ANGLE 360` or `EXPAND 1.001` (the latter will affect axis tick labels too). This implies (correctly) that the fonts are affected by the current expansion and angle. Another strategy is to start the string with a \0 to force an expansion factor of unity.

Now for old-style labels (which are still the default): Type α or $\backslash\alpha$ to change to font α for one character (first form) or permanently (second form). The possible fonts are `g`, `o`, `p`, `r`, `s`, and `t` for 'greek', 'old english', 'private', 'roman', 'script', and 'tiny' respectively. In addition, the pseudo-fonts `u` and `d` move text 'up' and 'down' respectively, and `i` produces 'italic' (actually just slanted) characters. Size changes are just like any other font change, so `\6` and `\\-4` will affect one character and the rest of the string respectively. This is really very much simpler than it sounds - try

```
label \gp\\u\\-21/2\\2\\d = \3\g:e\\u-x\u2\\d\s dx
```

Note that 'tiny' is a misnomer, it is (nowadays) just a font that look better if you need small letters (`\\t\\-6` will produce a shrunken 'tiny' font, just like the old days). Spaces are treated differently in different fonts, as a greek space is a negative space (*i.e.* a backspace), and a script space is only half as wide as a normal space.

The rest of this appendix is really for Gurus.

The characters in a font are specified using a programme **read_fonts** which you can use to make binary font files from the list of Hershey characters, using an index file to specify what character should go where. The binary fonts file also sepcifies which TEX 'definitions' are available (*e.g.* `\alpha`). The default font table is illustrated at the end of this appendix. Which font file you want to use is specified as the **fonts** entry in your **.sm** file. *. The **fonts.bin** fonts have been cleaned up a bit

* It's possible to resurrect the font table used by pre-2.0 versions of SM, using the index file **old_font_index**

for version 2.0 of SM, although the order of characters in the greek and roman fonts is unchanged. There is a new font, 'Old English' or \o or \oe, and a good number of new characters are provided. Neither of these fonts supports the 'private' font, that is there in case users desperately need something, when they can make their own binary font file. For example, there is a set of Hershey oriental fonts that could be used (we have it somewhere).

The complete list of (occidental) Hershey characters is given in a file called `hershey_oc.dat`, and is in the public domain. Each character is specified by a number in the first 5 columns, then a number of strokes in the next 3, then pairs of letters in the remaining columns up to 72, and in as many 72 character lines as are needed. (Annoyingly, if a line consists of exactly 72 characters, the next must be left blank). Each pair of characters consists of a coordinate, with the origin at (R,R), and the y axis pointing *down*. A ' ' indicates that the next point is a move, otherwise just connect the dots. The very first pair is different, as it specifies the left and right spacing for the character. If this isn't clear, try drawing a few characters on graph paper, character 2001 (roman A) for example. There are a few characters that have traditionally been available in Mongo that are not in the Hershey set, these have been added to the end of the **hershey_oc.dat** file, plus a few that we thought deserved adding.

If you want to create your own characters, the macro `make_char` in **fonts** (*i.e.* `load fonts make_char`) might help. It uses the cursor to make a string that is (nearly) in the correct form for inclusion in **hershey_oc.dat**

The programme `read_fonts` reads this file, an 'index' file that specifies the characters to be put into the fonts, and a list of TEX definitions. The index file consists of character numbers, or ranges consisting of two numbers separated by a minus sign. Comments go from the character # to end of line. Each font consists of 96 characters in ascii order, and fonts appear in the index in the order rm, gr, sc, ti, oe, and pr. The format of the TEX definition file is that each definition has a line to itself, lines starting with a # are comments. A line consists of a name, some whitespace, the name of the font to use, a single white-space character, and the value of the definition to the end of the line (or 40 characters, whichever comes first). For example

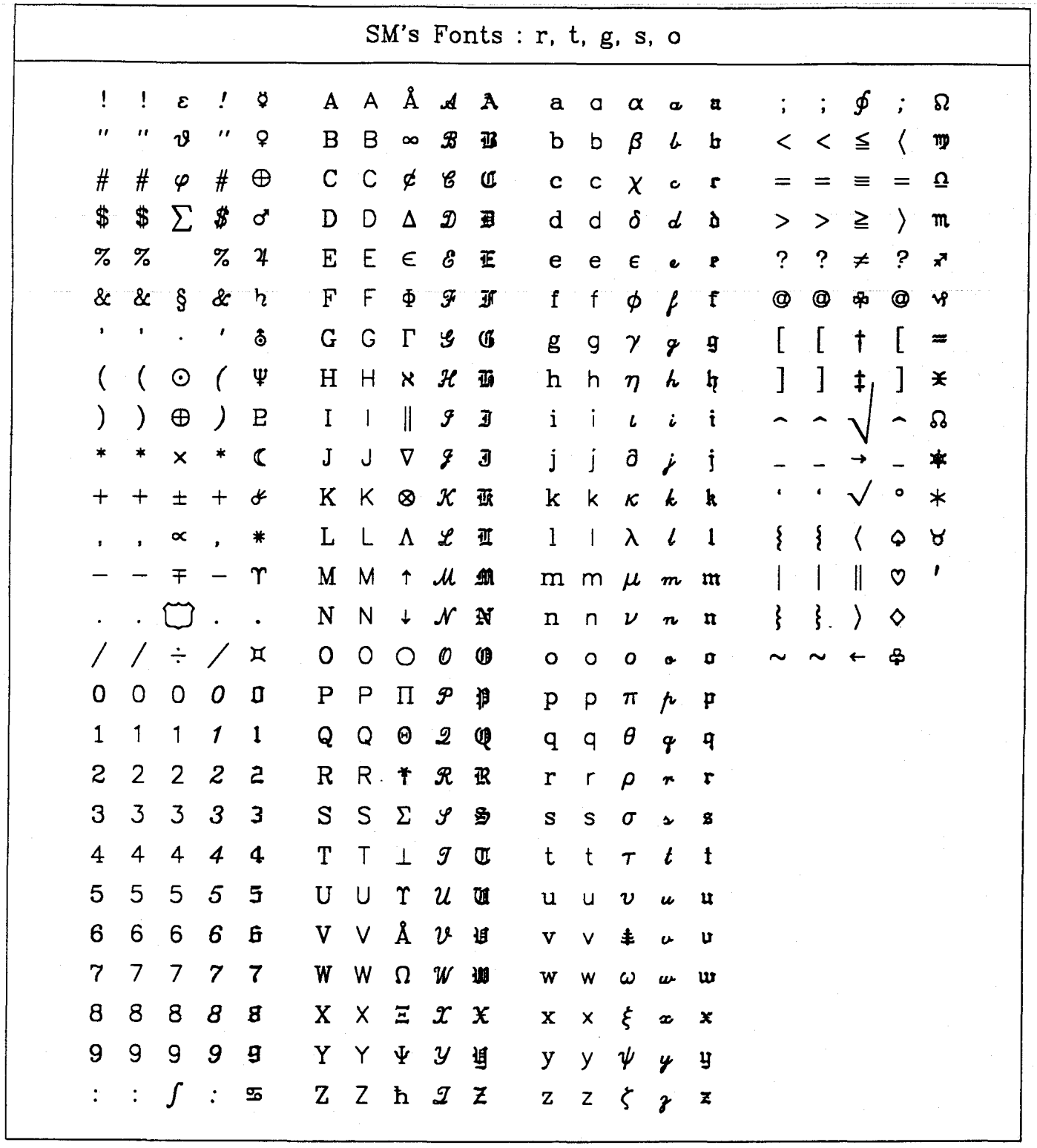
```
alpha          gr    a
alsoalpha      rm    \gr a
alphatoo       gr    \4 a
```

defines `\alpha` the conventional way (as the character `a` in the greek font), then defines `alsoalpha` in a less efficient way (by specifying the roman font, then explicitly switching to greek), then defines `alphatoo` as a big α. There's no reason why your definitions can't be reasonably complicated, see for example the definition of `\TeX`. The main Makefile prepares your binary font file for you.

SM's "TEX" definitions:

\. : Thin space
\! : −ve space
\' : ′
\\ : \
\AA : Å
\Alpha : A
\Aquarius : ♒
\Aries : ♈
\Beta : B
\Cancer : ♋
\Capricorn : ♑
\Chi : X
\Delta : Δ
\Earth : ⊕
\Epsilon : E
\Eta : H
\Gamma : Γ
\Gemini : ♊
\Iota : I
\Jupiter : ♃
\Kappa : K
\Lambda : Λ
\Leo : ♌
\Libra : ♎
\Mars : ♂
\Mercury : ☿
\Moon : ☾
\Mu : M
\Neptune : ♆
\Nu : N
\Omega : Ω
\Omicron : O
\Phi : Φ
\Pisces : ♓
\Pluto : ♇
\Pi : Π
\Psi : Ψ
\Rho : R
\S : §
\Sagittarius : ♐
\Saturn : ♄
\Scorpio : ♏
\Sigma : Σ
\Sqrt : √
\Tau : T
\Taurus : ♉
\Theta : Θ
\Upsilon : Υ
\Uranus : ♅
\Venus : ♀
\Virgo : ♍
\Xi : Ξ
\Zeta : Z
\aleph : ℵ
\alpha : α
\apoint a ns : Dot
\ast : ∗
\asteroid : ⚵
\beta : β
\bf : **Bold 'font'**
\bigcirc : ○
\cents : ¢
\chi : χ
\circ : °
\clover : ♧
\clubsuit : ♧
\comet : ☄
\dag : †
\ddag : ‡
\del : ∇
\downarrow : ↓
\diamondsuit : ♢
\delta : δ
\div : ÷
\epsilon : ϵ
\equinox : ♎
\equiv : ≡
\eta : η
\firtree : ⚸
\gamma : γ
\ge : ≥
\geq : ≥
\hbar : ħ
\heartsuit : ♡
\infty : ∞
\int : ∫
\iota : ι
\it : *Italic 'font'*
\kappa : κ
\kern dx : x += dx
\langle : ⟨
\leftarrow : ←
\le : ≤
\leq : ≤
\lambda : λ
\line lt len: Line
\mp : ∓
\mu : μ
\\nabla : ∇
\\ne : ≠
\\nu : ν
\odot : ⊙
\oe : Old English font
\oint : ∮
\omega : ω
\omicron : o
\oplus : ⊕
\otimes : ⊗
\palmtree : ✝
\parallel : ∥
\partial : ∂
\perp : ⊥
\phi : φ
\pi : π
\pm : ±
\point ns : Dot
\pr : Private font
\prime : ′
\propto : ∝
\psi : ψ
\raise dy : y += dy
\rangle : ⟩
\rightarrow : →
\rho : ρ
\rm : Roman font
\sc : *Script font*
\shield : 🛡
\sigma : σ
\snow : ❄
\spadesuit : ♢
\sqrt : √
\sum : ∑
\tau : τ
\theta : θ
\ti : 'Tiny' font
\times : ×
\uparrow : ↑
\upsilon : υ
\varepsilon : ε
\varphi : φ
\vartheta : ϑ
\xi : ξ
\zeta : ζ

SM's Fonts : r, t, g, s, o



Index

We have tried to make this index reasonably useful for all sorts of readers. We have avoided 'see something else' entries, at least partly because it was easier to write the index-preparation software that way.

.sm, 5.
TeX_strings, 128.
background, 50.
default_font, 129.
defining variables, 8.
file_type, 110.
fonts, 5, 130.
foreground, 50.
graphcap, 5.
history, 10.
history file, 6.
history_char, 14.
overload, 25.
save_file, 20.
term, 14.
ttybaud, 115.
2-dimensional graphics, 110.
contour, 44.
contour levels, 61.
cross section, 42.
file format, 110.
file_type, 110.
image, 58.
image max and min, 65.
3B1, device driver, 52.
C, interface to SM, 99.
GIN terminators, 46, 94.
I/O commands, data, 47.
image, 58.
lines, 62.
read, 70.
TeX, caveats, 130.
definitions, 129.
extensions, 129.
using in labels, 128.
Unix-PC, device driver, 52.
VMS, DCL escape, 14.
foreign command, 6.
keybindings, 5, 13.
names of callable functions, 99.
spawned processes, 5.
X10, device driver, 51.
X11, device driver, 51.
Xwindows, device drivers, 51.
angle, 40.
apropos, 40.
examples, 17.
arithmetic, 41.
operators, 42.
ascii labels, 68.
aspect, 41.
axes, drawing, 43.
grid, 56.
label format, 56, 66.
logarithmic, 76.
tick spacing, 76.
x label, 80.
y label, 80.
axis, 43.
bindings, names of operators. 12.
binning, see histogram, 42.
bison, copyright notice, 81.
box, 43.
calling SM, 99.
2-D example, 102.
2-D function definitions, 101.
Libraries, 99.
example, 99.
function definitions. 100.
case sensitivity, 5.
changing, meanings of commands, 24.
changing directory, 14, 44.
chdir, 44.
classifying strings, 78.
color, graphcap entry, 95.
used in plotting, 45.
colour, graphcap entry, 95.
used in plotting, 45.
command history, 57.
command line, -f option ignored. 96.
commands, 6.
environment file. 6.
logfile, 7.
macro files, 6.
specifying .sm file. 6.
stupid, 7.
suppress echo. 7.
verbose, 7.
comments, in data files, 70.
in macro files, 17.
in macros, 16.
compatible, with Mongo, 126.
concatenate, 22, 42.
connect, 44.
continuing long lines, 5.
contour, 44.
control words, apropos, 40.
chdir, 44.
define, 47.
delete, 49.

do, 53.
edit, 53.
else, 58.
foreach, 56.
help, 57.
history, 57.
if, 58.
key, 59.
listing, 62.
macro, 64.
overload, 66.
print, 67.
prompt, 67.
quit, 69.
return, 71.
set, 72.
show, 75.
termtype, 75.
user, 76.
verbose, 77.
version, 78.
write, 79.
ctype, 45.
current directory, 14, 44.
cursor, 46.
graphcap, 94.
customising SM, 27.
data, 47.
define, 47.
delete, 49.
macros, 16.
device, 49.
Unix PC, 52.
Xwindows, 51.
adding new ones, 95, 116.
stdgraph, 50.
sunview, 51.
sunwindows, 51.
diagnostics, verbose, 77.
do, 19, 53.
dot, 53.
double quote, 5.
draw, 53.
edit, 53.
bindings, 10, 12, 13.
commands, 10.
cursor being weird, 76.
editing a macro, 18.
escaping, 12.
history, 10.
padding, 115.
size of screen, 76.
termtype, 14, 75.
environment file, see .sm, 5.
erase, 55.
erasing lines, 64.
errorbar, 55.
errors, handler for, 4.
examples, 32.
basic, 2.
macros, 26.
parabola, 2.
exiting SM, quit, 69.
expand, 55.
fft, 56.
file_type, 110.
filecap, definition, 110.
examples, 110.
floats, as words, 82.
fonts, .sm, 5, 130.
available characters, 128.
choice, 128.
hardware character sets, 128.
foreach, 19, 56.
format, 56.
fortran, interface to SM, 99.
good advice, 32.
good ideas, 72.
grammar, character processing, 81.
do, foreach, and if, 84.
examples, 84.
macros, 83.
peculiarities, 82.
token generation, 81.
tokens, 81.
variables, 7.
verbose debugging, 84.
graphcap, .sm, 5.
compiling, 98.
description, 86.
entering nul, 87.
overriding on command line, 51.
raster devices, 96.
writing a new entry, 95.
grid, 56.
handler, see errors, 4.
hardcopy, example, 3.
help, 20, 57.
.sm, 6.
macros, 15.
see also apropos, 40.
variable, 20.
vector, 20.
histogram, 57.
binned from vector, 36, 42.
history, 57.
changing character, 14.

converting to a macro, 18.
deleting commands, 10.
editor, 10.
introduction, 9.
listing, 9.
listing backwards, 9.
number remembered, 10.
re-using commands, 9.
reading from a file, 71.
reading from a macro, 18.
identification, 57.
if, 19, 58.
image, 58.
initialisation, See startup, 6.
key, 59.
defining keyboard macro, 18.
label, 60.
labels, aspect ratio, 41.
identification, 57.
partly in different fonts, 61, 87.
landscape plots, 50.
levels, 61.
limits, 61.
line drawing, colours, 45.
line style, 64.
line weight, 64.
on a new device, 50.
lines, 62.
list, 62.
listing, key bindings, 12.
macros, 15.
variables, 8.
vectors, 24.
location, 63.
logarithmic, getting nice axes, 2.
logical operators, 63.
long lines, continuing, 5.
lost commands, 6, 10, 30.
ltype, 64.
lweight, 64.
macro, 64.
macros, .sm, 6.
arguments, 15.
comments, 16.
defining, 15.
deletion, 16.
disk format, 17.
editing, 18.
error_handler, 4.
from history list, 18.
help, 15, 16.
introduction, 15.
libraries, 121.
listing, 15, 16, 62.
not listing, 16.
reading from disk, 17.
returning from, 71.
saving to disk, 20.
to history list, 18.
undefining sets, 17.
usage, 15.
useful, 121.
variable number of arguments, 15.
writing to disk, 17.
minmax, 65.
mouse, buttons, 95.
name, .sm, 6.
notation, 66.
numbers, as words, 82.
overload, 24, 66.
paging, paging through lists, 20.
panic, save file, 20.
parenthesised expression, 44, 55, 56, 61, 67, 68, 74.
plot macros, 64.
writing to history list, 79.
plot size, plot window, 79.
to change, 63.
plotting commands, angle, 40.
aspect, 41.
axis, 43.
box, 43.
connect, 44.
contour, 44.
ctype, 45.
cursor, 46.
data, 47.
device, 49.
dot, 53.
draw, 53.
erase, 55.
errorbar, 55.
expand, 55.
format, 56.
grid, 56.
histogram, 57.
identification, 57.
image cursor, 58.
label, 60.
levels, 61.
location, 63.
ltype, 64.
lweight, 64.
minmax, 65.
notation, 66.
plot limits, 61.

points, 66.
ptype, 68.
putlabel, 69.
relocate, 71.
shade, 74.
sort, 75.
spline, 75.
ticksize, 76.
window, 79.
xlabel, 80.
ylabel, 80.
plotting devices, see device, 49.
points, 66.
plotting, 66.
point style, 68.
size, 55.
porting to new machines, 120.
portrait plots, 50.
precedence, arithmetical operators, 42.
logical operators, 63.
print, 67.
private initialisation, 27.
prompt, 67.
ptype, 68.
putlabel, 69.
quit, 4, 69.
range, 69.
raster, adding new devices, 97.
graphcap support, 96.
rasterise, description, 96.
read, 70.
macro, 17.
relocate, 71.
restore, 20, 71.
return, 71.
save, 20, 72.
scrolling, graph disappears off screen, 115.
set, 72.
shade, 74.
surprises, 74.
shading regions, 74.
shell escape, 14.
show, 75.
sort, 75.
special characters, arithmetic, 5.
backslash, 5.
caret, 5.
dollar, 5.
double quote, 5.
exclamation mark, 70.
hash, 5, 70.
square brackets, 37.
special symbols, plotting, 66.
spline, 75.
startup, command line, 6.
default, 6.
history file, 6.
macros, 6, 17.
private macros, 27.
system initialisation macro, 26.
startup2, private startup macro, 27.
string vectors, declaring, 73.
into variables, 18.
ptypes, 68.
reading, 70.
setting elements, 73.
strings, preserving quotes, 82.
sun, cursor, 46.
device drivers, 51.
syntax error, 4.
tracking down, 77.
termcap, description of file, 114.
used by termtype, 75.
terminals, description for SM, 114.
graph scrolling off screen, 14.
setting terminal type, 75.
specifying screen size, 14.
termtype, 14, 75.
text plotting, 60.
angle, 40.
aspect ratio, 41.
character size, 55.
putlabel, 69.
x axis label, 80.
y axis label, 80.
ticksize, 76.
upc, device driver, 52.
uppercase, making SM understand, 5.
user, 76.
variables, accessing internal, 8.
concatenation, 8.
defining, 47.
deletion, 7.
environment, 54.
forcing expansion, 8.
from .sm, 8.
from data files, 8.
from internal values, 8.
from keyboard, 7.
help, 20.
introduction, 7.
listing, 8, 62.
mlist, 27.
quotes, 8.
saving to disk, 20.
special values, 8.
syntax, 7.
temporary, 16.

testing if defined, 8.
use, 7.
vector field, 55.
vectors, arithmetic, 42.
concatenating, 22.
declaring, 23.
defining, 72.
extract histogram, 42.
help, 20, 23.
listing, 24.
logical operators, 63.
printing, 67.
saving to disk, 20.
sorting, 75.
spline fitting, 75.
verbose, 77.
variables, 7.
version, 78.
weird, Can't enter superscripts, 130.
alpha cursor being, 14, 76, 115.
behaviour of keys, 11, 13.
can't get prompt, 4, 9.
commands misbehave, 21.
commands redefined, 21.
delete in foreach, 19.
grapcap eats a \377, 87.
losing ends of macros, 71.
losing hardcopies, 71.
lost arguments to macros, 16.
lost commands, 6, 10.
won't plot log axes, 62.
wrong macro in message, 1.
whatis, 78.
window, 79.
write, 79.
macro, 17.
to a file, 79.
to the terminal, 79.
xlabel, 80.
xwindows, device drivers, 51.
ylabel, 80.

A SM Tutorial

1. What is SM?

SM is an interactive plotting package for drawing graphs. It does have some capability to handle image data, but mostly works with vectors. The main features of the package are that one can generate a nice looking plot with a minimum number of simple commands, that one can view the plot on the screen and then with a very simple set of commands send the same plot to a hardcopy device, that one can build and save ones own plot subroutines to be invoked with a single user-defined command, that the program keeps a history of ones plot commands, which can be edited and defined as a plot subroutine, to be reused, and that one can define the data to be plotted from within the program, or read it from a simple file.

2. Why do I need SM?

I am not going to answer this question. If you read through this tutorial, and use the package for a while, and still can't see why you need it, then you probably don't need it.

3. How to get data into SM

Plot vectors may be generated in several ways

a. You may read the vectors from a file using the **read** command The file is expected to be an ASCII file of columns of numbers (separated by spaces or commas). You define the file to SM using the **data** command, and associate a column or row of numbers with a SM vector using the **read** command. Example: Say I have a file named test.dat with the following data in it:

```
1  2  3    5.610
3  6  8    2.311
5  8  2    7.712
7  9  4    9.313
9  3  1    4.814
```

Then the commands to issue to SM to get the data into the program are:

```
data test.dat
read x 1
read y 2
```

In the last 2 commands I have told SM to read the values in column 1 of the file test.dat, and assign them to a vector named x, and read the values in column 2 of the file and assign them to a vector named y. I could read any of the other columns in as well, of course, and assign them to vectors. And I can name the vectors whatever I like, as long as the name consists of the characters a-z,A-Z,0-9, and _ (underscore). I can also read a row from the file, instead of a column, by saying

```
read row x 1
```

Note that the vector is defined by the read command. But I can redefine it whenever I wish, and change the size. The only point to remember is that when you redefine the vector, the old values are overwritten.

A final point to note about defining vectors from files is that you can skip over lines in the file with the **lines** command. **lines** defines which lines in the file you want to read. A limitation of **lines** is that you may only define one set of lines to read; that is, if you had a 30 line file, and wanted to read lines 3-9 and 15-30, you couldn't (well, you could, but you'd have to make

clever use of the method of defining vectors which is discussed under number 2 in this section, or make lines 10-14 each begin with a #).

b. You may define the vectors within SM using the **set** command. This command has a number of forms:

- If you just want to define the vector with a list of values, the command is

```
set numlist = { 2 3 4 5 6 7 8 9 27 }
```

- you can also define a vector in terms of arithmetic operations on a previously defined vector. For example, having defined numlist as above,

```
set ylist = sqrt(numlist) + numlist/3.1
```

the allowed arithmetic operators are **sin, cos, tan, acos, asin, atan, abs, int, lg, exp, ln, sqrt, concat, **,+, -, *,** and /, where **lg** is $\log_{10}$, **ln** is $\log_e$, **int** takes the integer part, and concat concatenates two vectors.

- you can define a vector with an implied do-loop:

```
set x = 0,PI,PI/16
```

or

```
set y = 0,10,0.1
```

(PI is a constant defined for you by SM, but you can also define your own scalar variables, as will be described in a later section of this tutorial)

c. you can redefine an *existing* vector element by element with a do loop:

```
set y = 1,50
...
do i=0,100,2 { set y[$i] = $i**2}
```

(*Note* that vector elements are numbered starting from **0**)

d. You may create a vector with the **spline** command. This fits a spline function to a previously defined pair of vectors, and evaluates it at the points given in a third vector, to produce a fourth vector for you.

```
set x = 0, 2 * PI, PI/4
set y = sin(x)
set xx = 0, 2 * PI, PI/32
spline x y xx yy
```

This will fit a spline to the curve y vs x, at the points 0, PI/32, PI/16, 3 * PI/32,... (i.e. the points in the xx vector), and the spline values will be stored in the vector yy.

e. You may define a vector with the graphics cursor using the **cursor** command. If you type the command

```
cursor a b
```

then the crosshair cursor will be displayed on the screen, and to the **spline** command. This will take a horizontal slice through the image. If you do not give a filename, the vectors are printed to the terminal.

4. How do I generate a basic plot for data in a file?

The procedures listed above describe how to get your data in to SM. Then the steps to plotting it are as follows:

- Define the plot device you want to use with the **device** command. The syntax and list of available devices are described in the manual.

- Declare the data file with the **data** command, as described in part 1 of this tutorial (and in the manual). You might want to use the macro **da** instead, which stops SM worrying about things like **$** or **/** in the filename.

- Read in the data vectors with the **read** command, also described in part 1 (and in the manual)

- Define the axis limits with the **limits** command.

- Draw the axes with the **box** command

- Plot the data points using the **connect** command to plot the points as a connected curve, or the **points** command to plot them as points. If you are using **connect** you can also define the line type with the **ltype** command (see the manual for the available line types) If you want to plot points, you should first define the point style with the **ptype** command. **ptype** allows you to define the point as a type of polygon, with any number of sides, and 4 basic forms (it is described in the manual). Ptype also allows you to define your own private symbols, or use strings that you've read from a file. You can also plot characters, or character strings in the point location, by defining another vector that contains the characters you want to plot, and using the **alpha_poi** macro (see the manual for a description and example, in the section on useful macros)

- If you want to label the axes, use the **xlabel** and **ylabel** commands. SM is able to plot Greek letters, superscripts and subscripts, many sorts of symbols, and a couple of other special fonts. The available fonts are listed in the manual in Appendix IX.

There are a number of other commands that elaborate on this basic set to allow logarithmic axes, labelling curves, putting an ID on the plot, reading positions from the plot with a cursor, manipulating 2-D data, and much more. Some of these commands are described in section 12; the definitive source, however, is at the back of the manual, where ALL the commands plus their syntax are described.

5. What do I have to do to start up SM?

With any luck, your system manager has set up SM so that you can run it by typing a single command. But in addition to this, you need to have a file in your home directory named **.sm**. This file is used by SM to tell it a number of things. A prototype **.sm** file is reproduced below:

```
device hirez
edit $disk:[sm_dir]maps.dat
filecap $disk:[sm_dir]filecap
file_type vms_var
fonts $disk:[sm_dir]fonts.bin
help $disk:[sm_dir.help]
history 80
history_file .smhist
graphcap $disk:[sm_dir]graphcap
macro $disk:[sm_dir.macro]
```

```
macro2 $disk:[lupton.sm]
name my_friend
printer qms
temp_dir □
term hirez
termcap $disk:[sm_dir]termcap
```

Now, what is all this?

device allows SM to initialize a default plotting device for you. If it finds an entry of this type in your **.sm** file, it will do the call to the **device** command for you.

edit is used by SM to find out how to map the key sequences used by the macro and command line editor. The only reason you need to change this line is if you decide you want to define the key sequences differently. For example, if you invoke the macro editor (see section 8 to find out what that means), and want to edit your macro and, say, insert a line, to do that you type CTRL_M. Now, if you don't like that particular choice of keys, you can set up your own key definitions in your own edit file, and tell SM to use that instead of the default ones by redefining the **edit** line of the **.sm** file.

filecap is a file that tells SM how to read 2-dimensional files. As mentioned in section 11, binary files vary enormously from operating system to operating system, and also depend on the language of the program you wrote to generate them, so we defined a few simple file formats you can use to read binary data into SM, and SM interprets them via the **filecap** file. Read the manual if you want to plot 2-D data.

file_type is also for 2-D data. This sets the default file type for the binary files. The types are described in the manual. The **filecap** file tells SM how to read data of the given **file_type**

fonts This tells SM where to find the font definition file. You will almost certainly never change this, but if you have made a new font file you would cause SM to use it instead of the one we supply by changing this line in your **.sm** file.

help This tells SM where to find the command help files.

history The number of history commands to remember (see the next entry).

history_file SM keeps of history of the commands you used in your SM session in a file. It reads in the last history file when you start it up again, and you can reuse those commands as you wish (e.g. scroll through them like with the VMS command line editor, extract a group of commands into a macro (see section 8),...). If you don't want a history, leave this line blank. Otherwise, specify a filename.

graphcap This is the file SM uses to figure out what magic commands to send to your plot device to cause it to go into graphics mode. We have defined many device types, so hopefully the one you need is already in the default file. If not, you may want a private graphcap.

macro SM loads a set of default plot macros for you when you start it up. This line gives the location of the default macro file.

macro2 You can load 2 default files if you wish, and this is where you define the second one. **macro2** is the name of a directory where SM expects to find a file name **default**, in which are contained SM macros. You should load our default one first, for reasons which are explained in the manual. The macro **startup2** in file **default** will be executed.

name This is the name by which SM will address you when you use it.

printer There is a macro called **hcopy** that replays the commands used to generate a plot on your screen and changes the device to a printer to allow you to easily get a hardcopy of your screen plot. This line tells SM what printer you want to use. You can also get hardcopy plots manually (see section 10)

temp_dir Hardcopy plots are written to a disk file, and then submitted to a print queue and deleted. This tells SM what directory you want it to write the disk files to. They can be large, so if you have disk quota problems, **temp_dir** ought to point to a scratch disk or something

term SM knows about terminals, and uses that knowledge to allow you to do command line editing. Here is where you specify what kind of terminal you have. Note this is for **text** only; the graphics description is given in the **device** line at the start of this file. The available terminal types are described in the **termcap** entry

termcap This file describes terminals.

So to run SM, you should have a file like this in your home directory, with the directory names, etc changed to point to your computer and you, and then just run the program. If all goes well, when you invoke the program, you will wait a while, and then get the following message

Hello, <**name**>, please give me a command

where **name** is as defined in the **.sm** file **name** line, and you will get a prompt. If this isn't what happens, you need to contact the people who installed SM on your system.

6. How do I define variables, and where and how can I use them?

Scalar variables are defined with the **define** command. As mentioned above, vectors are defined with the **set** command. A variable may be a number, or a character string. You may use them in any SM command, by preceding the name of the variable with a $. For example:

```
define 2PI 6.283
set x=1,100
do i = 0, 2, .01 {
set x[i] = i * $2PI }
set y = sin(x)
limits x y
box
define xlab { my signal }
xlabel $xlab
ylabel sine
```

7. What is a plot macro, and how do I make one?

A plot macro is a set of commands that you can execute together by invoking the name of the macro; in effect, it is a plot subroutine. For example, suppose you had a set of plots that you wanted to generate, using the same type of axis box and labels. Rather than laboriously typing the box and label and limits commands for each set of data, you could define a macro as follows:

```
drawbox
    limits 0 20 0 100
    box
    xlabel xdata
    ylabel ydata
```

where the macro name in this example is **drawbox**. Then, when you access your data, you could do as follows:

```
data file1.dat
read { x 1 y 2}
drawbox
connect x y
data file2.dat
read x 3
read y 7
drawbox
connect x y
```

(The **read { x 1 y 2 }** is the same as **read x 1 read y 2**, but faster). This is a simple-minded example, and you can immediately see ways to improve the macro I have created to save even more typing. Macros may consist of any SM commands, and may have arguments. You specify the number of arguments in the macro definition, and refer to them by number, preceded by $. In the example I gave above, suppose we wanted to make the axis labels into variables. Then the macro definition would look like this:

```
drawbox 2
    limits 0 20 0 100
    box
    xlabel $1
    ylabel $2
```

Then to invoke the macro, I type

```
drawbox xdata ydata
```

You can make a macro in 4 ways:

a. You can create it with your favorite editor outside of SM. The rule to remember if you do this is that the name of the macro must be the first thing on a line of the file, and should be followed by SM commands. All the commands must start in a column in the file other than the first column. To read this macro into SM, use the **macro read** command

```
macro read macro.file
```

will read all the macros in the file **macro.file**.

b. You can define the macro within SM by extracting a set of commands from the history buffer

```
macro mname 1 20
```

will extract lines 1 through 20 from the history buffer, and create the macro **mname** which consists of those 20 lines.

c. You can define the macro within SM with the **macro** command

```
macro mname {
```

will start the definition of the macro named **mname**. You then enter SM commands, and terminate the macro definition with a closing }.

d. You can define the macro within SM with the **macro edit** command

```
macro edit mname
```

will invoke the macro editor, and you can then enter SM commands to define the macro. The editor is described in detail in the SM manual; the main commands to remember are as follows:

- the editor is a line editor, not a screen editor. You can advance to the next line with the down arrow on the keyboard, and up to the previous line with the up arrow. Similarly, the right and left arrows advance the cursor one character right and left, respectively.

- within a line, keys will work just like the history editor

- to insert a line above the current line, type CTRL-O

- to insert a line just after the cursor, type CTRL-M

- to erase to the start of the current line, type CTRL-U

- to advance to the end of the current line, type CTRL-E

- when you type into an existing line, this acts in insert mode, not overwrite mode. To overwrite an existing character, position the cursor just after the character you want to overwrite, use the delete key to erase the character, and then type in the new character (or lookup how to set overwrite mode in the real manual)

- to exit the macro editor, and save the changes, type CTRL-X.

8. How do I save macros?

Once you have defined the macro, the command

```
macro write macro1 macro_file.dat
```

will write the macro named macro1 to the file macro_file.dat The **macro write** command remembers the name of the last file it wrote a macro to, and if the filename is the same in the next command, it will append the new macro to the file, otherwise it will delete it first (you can get round this – see the manual). In this way, related macros can be written to the same file.

Another (maybe easier?) way is to use the save command. the command

```
save save_file
```

will save everything to a file – macros, variables and vectors. To get them all back, say

```
restore save_file
```

You can even logout, go to dinner, come back, restart SM, use restore, and be back where you left off.

9. How do I get a hardcopy of a plot?

You can simply define the hardcopy device with the **device** command, then issue the plot commands, and then type

```
hardcopy
```

which sends the plot to the hardcopy device.

Or, in the more common scenario, you have put the plot on the screen, and fiddled with it until you were happy with it, and then want to plot it to a hardcopy device. In this case, you make use of the fact that SM saves your plotting commands in a buffer, and you can manipulate that command list. The command

```
history
```

will print out the list of commands, in reverse chronological order (or chronological order with **history -**). You can then delete all the commands in that buffer that you don't want with the **DELETE** command.

```
DELETE 1 10
```

will delete lines 1 through 10 from the history list. Once you have deleted all the lines from the history list except the ones you used to make the plot on the screen, you can change devices to the hardcopy device using the **device** command, and then type

```
playback
hardcopy
```

This will execute the commands in the history list, and then print the hardcopy plot. In fact, there is a macro **hcopy** defined to do this for you. **hcopy** sets the device to the hardcopy device (as defined in your **.sm** file on the **printer** line), then does a **playback**, then sends the plot to the hardcopy device, and then resets the device type to be whatever it was when you invoked the **hcopy** macro. You don't have to playback all the lines; both **hcopy** and **playback** have optional arguments to specify the range of lines that you want.

You could also define the commands from the history list into a macro, as discussed in section 6, and invoke the macro to execute the plot commands. **macro hcplot 1 20 device qms lca0 hcplot hardcopy** This will execute the plot commands from the history buffer lines 1 through 20, and then send the plot to the hardcopy device for printing.

An important point to note about the hardcopy devices is that you have to reissue the **device** command each time you do a **hardcopy** command. This is because the hardcopy plot vectors are actually written to a file, and this file is closed, sent to the plotter, and deleted when you issue the **hardcopy** command. No new file is opened for you automatically, so you must issue the **device** command to open a new plot file if you want another hardcopy plot, or to redefine the device to a terminal, if that is what you want.

10. What about 2-dimensional data?

SM has some capability for handling image data. You can define an image with the **image** command, which is analogous to the **data** command for vectors. As described in the manual, you must first tell SM what sort of image file it is. Binary data is rather tricky to define in a general way, and certainly differs from one operating system to the next, so the few standard types of binary files we have defined will hopefully cover most cases, and if not, you can always write a program to convert your data into one of those types, or try to teach SM about your data format after reading the filecap appendix to the manual.

Once you have read in the image, you can contour it with the **contour** command (first define the contour levels with the **levels** command), and you can take a slice through it with the **set x = image(x,y)** command. More capabilities will probably be added, someday, but SM is not intended to be an image processing system.

11. What are the other common commands?

- Perhaps the most important thing to note in this context is that SM has a command line editor, which allows you to recall previously typed commands (use the up arrow to scroll back through them) and either reexecute them, or edit them. For those of you familiar with VMS, the command line editor is very similar to the one provided with that system.

- The next most important thing is that there is a CTRL_C trap in SM, so if you start a command and regret it, you should be able to abort it by typing CTRL_C (hold down the CTRL key and then press the c key)

- If you don't like the axes drawn for you with the **box** command, you can tailor them a bit more with the **axis** command.

- **cursor** invokes the device cursor (for devices that have one). You can then read positions from the screen by positioning the cursor, then typing any key except **e** or **q**. Those latter 2 keys are used to exit the cursor routine.

- **end** causes SM to exit.

- **expand** changes the size of the points and characters drawn on the screen, as well as the size of axis tickmarks.

- **format** allows you to specify the format of the numbers that are plotted along the axes.

- **help** is a very important command. You can also specify help on a particular command with **help <command_name>**

- **identification** plots an identification line at the top of the graph, giving the date and some other information, of your choice.

- **label** allows you to plot a label on the graph, at the current location. You can specify different fonts and symbols, as described in the manual.

- You can change the size of the plot window with the **location** command. But in general, if you want to plot more than one graph on the screen (or page) at once, you will probably use the **window** command (q.v.)

- **lweight** allows you to change the line thickness. This will also apply to characters that are plotted.

- By default, if the numbers you plotting on the axis are between .0001 and 10000, SM will write these numbers out in decimal format. The **notation** command allows you to specify the range of numbers that you want written out in this way, as opposed to being written out in exponential notation.

- **quit** causes SM to exit.

- **relocate** relocates the current plot position to wherever you specify when you issue the **relocate** command.

- **ticksize** is used to control the spacing of tickmarks on the axes. Its most common use is to define a logarithmic axis. To do this, the first and third argument to the ticksize command should be negative.

- **window** is used to draw more than one graph on a single screen (or piece of paper). As the name implies, it divides the default plot window into n by m subwindows. You can make the windows touch, if that is what you want to do.

There are many more commands, which are described at the back of the manual. You will regret not reading about them.

12. What are the likely sources of error, and what do I do about them?

- If you try to run the program, and it says it can't find graphcap, or font, or edit files, you probably don't have a `.sm` file. If you do have one, it must be in your main directory. If it is there, it must have been edited to look for the files in the correct place, and not be trying to read them from Baltimore. If the directory specifications are correct, the files probably have not got the correct access permissions set, so whoever installed SM should fix that.

- If you try to plot a vector that you read from a file, and it says the vector is not defined, it probably means that the file has some non numeric stuff in it, that you didn't skip over with the `lines` command. If not, we have found problems in some cases with SM trying to read a file written by a VMS Fortran program. If yours is such a file, just use the VMS editor to make a new copy of the file, and it should be ok. The thing to look for is whether the file has Fortran carriage control attributes (do a `dir/full` command on the file, and it will tell you). When you edit it and make a new copy, those attributes will be replaced with normal ones. It does read most Fortran files, and we are not sure how the ones it can't read were written.

- If you make a syntax error, SM will tell you so, and reprint the line with a little arrow indicating the point in the command at which it got confused. One place that a syntax error is apt to arise is if you make use of the ability of SM to accept more than one command on a line. Certain commands cannot be used in this way, because it is ambiguous to the parser what you meant. This is described in the manual in Appendix I in the section entitled Peculiarities of the Grammar.

13. Where do I go from here?

To the real manual, of course, wherein you will find all the commands described, a list of all currently defined macros and what they do, plus a description of the program structure as well as information about the graphics back end that will enable you to add drivers for other devices.

To: New SM Users SM From: Robert Lupton

This is the third-and-a-half distribution of SM *, but it might still have a few problems. For this release (2.1) we have taken the opportunity to officially change the Programme's name to SM, which is what we ourselves have called it since its early childhood. For users this change will manifest itself in a couple of ways: the executable has a new name (**sm** or **sm.exe**), the environment file has a new name (**.sm**), and the source directories have a new name (**sm**) which may (or may not) require changes in **.sm** and **graphcap files**. We apologise for any inconvenience, of course, but there were and are good reasons for the change. As usual, the authors are actively using SM and it is pretty reliable. (If you don't agree send mail! You may be able to do this with the `gripe` command).

First a note about compiler-compilers. The SM command interpreter is written in YACC ('Yet Another Compiler-Compiler'), which is a Unix utility which converts a set of rules into compilable code. Unix is of course not in the public domain, but fortunately there is a version called 'Bison' written by the the Free Software Corporation, the people who wrote Gnuemacs. As far as I know, it doesn't matter which you use if you have a choice. So that is what the cryptic comments about Yacc and Bison are about.

Setting up SM is slightly dependent on what operating system you are using. The following two paragraphs deal primarily with Unix. Then there is one dealing with the differences under VMS, but otherwise you are on your own. Look up 'porting to new machines' in the index of the manual if you want help with getting SM running on a new system.

Unix first: Use the provided Makefile to make the executable - `make all` in the main directory. It might be wise to run `make empty` first, in case there are unsuitable object files on the distribution tape. As distributed we assume that you will be using Bison not Yacc as your compiler-compiler, but it should make no difference. If you prefer to use Yacc, edit the Makefile in the src directory to remove all of the references to Bison, and all should be well. You should edit the file **sm/src/options.h** to select the options that you want. You may have to play with some of the options such as `NO_ALLOCA` before SM will compile at all. Then copy the executable to a suitable place (using `make install`, if you wish, which also installs filecap, graphcap and the font binary files. The 'suitable place' is defined in the Makefile). As distributed the code is compiled with the debugger and no optimisation, but you can change this in the Makefile. If you want to compile in some display devices, read the stdgraph appendix in the manual and act accordingly. Then edit the **.sm** file so that it points to the help and macro directories (initially subdirectories of the main source directory), and to the graphcap and fonts.bin files (initially also in the main directory). There is a skeleton **.sm** file provided, called unix.sm, which should

* Why is it called SM? There is a package called Mongo with a similar command syntax that is in common use in Astronomy departments around the country. Mongo was used as a model for SM, and that's where the 'M' came from.

be edited, and renamed **.sm**. Then copy the resulting file to the home directory of any prospective user, or they could use the -f command line flag to specify it, *e.g.* `sm -f /dd/sm/.sm`. It is essential that all users have a **.sm** file.

You are now ready to start SM for the first time, so execute the file `sm`. Try `load demos square 20` after selecting a graphics device with the `device` command, *e.g.* `device tek4010`. It would be as well to run `LIST DEVICE` or to look at the graphcap file and see which entries are useful to your users as they are sure to ask. If you can't decide which to use, experiment. The same thing goes for terminals. Here the only problem is with cursor motions, some terminals work better with cursor motions disabled (*e.g.* `TERMTYPE selanar -21` for some flavours of graphons). Before unleashing SM on the general public, you might want to edit the macro `startup` in the file `default` in the macro directory to add local comments/warnings.

Under VMS things may be a little nastier. If you have bought a C compiler and MMS, then use MMS to recompile the code as described for unix. Failing MMS, but with a C compiler, use the command file `compile.com`. Look at the header of this file to see how to use it, you have to define a logical `CURRENT`, then run `@compile all`. Then edit the **.sm** file as before, starting with vms.sm, and choosing your favourite scratch space. If you don't specify a directory for `temp_dir`, SM will use the current directory. You may have to increase some quotas to use SM - the problem will show up when you attempt to produce a hardcopy of a plot. You may need to increase the number of allowed sub-processes, you will need at least 2 subprocesses, or 3 if you run SM in a spawned subprocess as we recommend. When all else is ready, define a foreign command symbol to run sm.exe, and another to run rasterise.exe if you have any raster devices. Then proceed as before. Because SM uses the 'termcap' file invented by Unix, you'll need a copy yourselves. There is a small one in the main source directory and you'll have to make sure that either there is a termcap entry in the **.sm** file pointing to it (better), or a logical variable.

Problems

The worst sort of bug is the sort that leads to a core dump/traceback map. If this ever happens to you, **please** note down carefully what you were doing. If you are running Unix, please use dbx to examine the core file, and use the `where` command; under VMS please note down the stack trace. If dbx refuses to cooperate, please use the script 'dump' in the main directory, which uses adb to get at least some information. Then send the information to Robert Lupton or Patricia Monger. If you can repeat the problem, use SAVE to save your killer programme, and send it to one of us. Then we'll be able to `RESTORE`, `playback`, and see the problem for ourselves. † If you don't complain, bugs won't get squashed.

The next class of problems are things that you don't like, or things that you would like to be able to do. Incomprehensible manuals come under this heading

† User monger on scivax:: (on SPAN), or radio.astro.toronto.edu, or utorphys.bitnet, or user lupton on uhifa.ifa.hawaii.edu. If you don't like computer mail, send a letter to Robert Lupton, Institute for Astronomy, 2680 Woodlawn Drive, Honolulu, HI 96822

too. Please send them to us as well, and be as specific as possible. Please don't just say 'the manual was awful' or or 'Reading the manual was like my first exposure to James Joyce', but rather, 'I found the manual hard to get into. More explanation of how to manipulate files of macros would have been useful, and more examples', or 'I took 3 weeks to find how to draw logarithmic axes'.

Bug Fixes and Changes

These the changes are introduced in version 2.0, and refer to changes since version 1.2. Refer to file dist_1_2.tex for earlier changes.

- Fixed unix version of SM to accept redirected input. Who knows for VMS?
- Command line is read differently. The old behaviour (read one argument, treat it as a macro file, read macro file, strip path/suffix and attempt to execute) is preserved as the -m option. Otherwise, simply execute command line after the startup macro
- Added logfile option to command line (-l logfile), mostly for debugging
- Allowed `FOREACH var { list1 } { list2 }`, to allow keywords to appear in `list1`
- Fixed Interrupt handler to return control to user if requested
- Made spaces (and tabs) within quotes non-special, so they can appear as parts of single words
- New utility get_grammar written in C to extract grammar from yacc files
- Added support for ~ in CHDIR
- Made default verbosity 1 not 0
- Made PRINT page if printing to terminal
- The header lines written by PRINT now start with a #, so they can be read without using the LINES command
- Added commands SAVE and RESTORE to save and restore a complete session
- Replaced SHOW by a macro
- Fixed READ OLD to change ERRORBAR and WINDOW to new formats
- Allowed the use of \ to continue long lines
- Columns of data may now be separated by commas as well as white space
- Removed all restrictions on the number of columns in a data file
- HELP gives the values of variables and vectors, as well as macros
- In the unlikely event of SM detecting an internal error (e.g. a segmentation violation) a SAVE file called **panic.dmp** is now written to your current temporary directory

Bug Fixes and Changes

These the changes are introduced in version 2.1, and refer to changes since version 2.0. First real changes (as opposed to enhancements):

- The programme's name has been changed to SM, that the default directory name is now **sm**, and that the **.smongo** file is now called **.sm**.

- The compiled graphcap entry format has been changed (fixed); you must rerun **compile_g**.

- The command SET `name = macro(vector)` has been changed. Previously only one argument was permitted, but it could be an expression, and the result was returned by assigning to `$1` in a rather confusing way. Now many arguments are permitted (but they can't be expressions), and the result is returned as `$0`. See SET in the manual for more details.

- the SET DIMEN(`vec`) = expr statement has been altered, so the right-hand expression must be a number (with an optional suffix to denote a string-valued vector).

- The way you specify limits for logarithmic axes to AXIS is now consistent with the way you specify them to BOX.

- The 8th bit of graphcap entries is no longer stripped. To include an ascii null (`\000`) in an entry specify `\377`, a `\377` may be written `\377\377`.

- You must now include the library utils.a in addition to plotsub.a and devices.a when linking to SM as a graphics library.

Note that the programme's name has been changed to SM, that the default directory name is now **sm**, and that the **.smongo** file is now called **.sm**.

Changes to do with making SM

- Compilation options are now in a file src/options.h rather than in the Makefiles.
- The compiled graphcap entry format has been changed (fixed); you must rerun **compile_g**.

Changes to do with history and the editor:

- Syntax errors now report the macro in which they occur. Beware that this may be incorrect, if the macro has already been fully scanned.
- If VERBOSE is 1 or more, report the line that arithmetical errors occur on (*e.g.* mismatched dimensions, lg(0)).
- You can now choose your history character (default: ^). This may be useful for TEX strings, I use '".
- You can go to the start or end of a macro/the history list with ESC-< and ESC->. Use ESC-g to go to a line number.
- You can no longer type a space after the history character to insert it into a line – use quote_next instead (*e.g.* ESC q ^).
- Lines starting with a # in READ EDIT files are comments
- In **.sm** files comments run from a # to the end of the line.
- The various DELETE HISTORY commands now have siblings that omit the HISTORY and do *not* themselves appear on the history list.
- The way that DO, FOREACH, IF, and MACRO command lists are treated has been fixed. If VERBOSE is 5 or more such lists will be printed as they are scanned.

Changes to do with macros:

- The command APROPOS lists all macros whose names or initial comments match a given pattern.

Changes to do with variables:

- Variables given in <> or { } are no longer tokenised, so spaces will no longer appear in funny places.

Changes to do with vectors:

- You can now specify a printf-style format string to PRINT

- *The command SET* `name = macro(vector)` *has been changed.* Previously only one argument was permitted, but it could be an expression, and the result was returned by assigning to `$1` in a rather confusing way. Now many arguments are permitted (but they can't be expressions), and the result is returned as `$0`. See SET in the manual for more details.

- It's possible to FFT vectors (actually pairs of vectors for the real and complex parts).

- String valued vectors have been added to SM. They may be read from files (either by the row or by the column), printed, assigned to, tested for (in-)equality, and used to set the PTYPE. In the latter case the strings are used to label the points.

- the SET DIMEN(`vec`) = expr statement has been altered, so the right-hand expression must be a number (with an optional suffix to denote a string-valued vector).

- You can now define variables from IMAGE files (currently only supported for FITS), using DEFINE variable IMAGE

- The expression `vector[expr]` has been generalised to allow `expr` to be a vector, so the result is now a vector of the same dimension as `expr`.

- You can convert an expression into a string using `STRING(expr)`.

- It is now possible to specify that IMAGE files have no header, in which case you'll be prompted for the values of nx and ny (file_type no_header).

Changes to how plotting is done are as follows:

- The values of `TICKSIZE` now make a difference for logarithmic axes.

- The way you specify limits for logarithmic axes to AXIS is now consistent with the way you specify them to BOX. *This Is A Change!*

- GRID now takes an optional second argument, specifying only an x or y grid

- LTYPE ERASE allows you to erase some lines in a plot without erasing everything. This is only supported by some devices

- ANGLE and EXPAND can now be given as vectors, to specify different angles and expansions for each point plotted

- DOT will now work with user-defined ptypes

- You can specify a PTYPE of a vector of strings.

- The order of the default CTYPEs has been changed; you can now restore the default using the macro `reset_ctype`.

Changes to labels:

- TEX strings now makes `A_a^b` put both sub- and super- scripts on the same symbol (use `A_a{}^b` if that isn't what you want)
- A new bold (over-struck) `\bf` font is available.
- If you define the variable `default_font` it will be used as your default font for labels (including for axes). You can define it in your **.sm** file if you so desire.
- The position of the symbol for the constellation Taurus has been moved; this will not affect TeX_strings users. An Old English full stop is now a full stop.
- 'TEX definitions' are now specified in a file (and put into fonts.bin). This will not affect users, but does allow a SM administrator to add new definitions.
- You can use `\def` to define your own TEX definitions.
- You can specify more than 9 sides in TEX 'point' or 'apoint'
- The 'TEX' commands `\raise` and `\kern` are now available to move parts of strings about.
- TEX-labels can now be put into variables without difficulty.

Changes to stdgraph:

- A new encoder command `'str'` prompts you, then reads a character from the keyboard.
- You can specify graphcap entries (or parts of graphcap entries) on the command line.
- The 8th bit of graphcap entries is no longer stripped. To include an ascii null (`\000`) in an entry specify `\377`, a `\377` may be written `\377\377`.

Changes to drivers

- You can now specify options like `-Ws` to the sunview driver, use `-WH` for a list.
- There is a new driver for a Unix-PC.
- There is a new X11 driver (try DEVICE x11 -help for a list of options).
- If you write your own drivers, the declarations of some functions have been changed to return void.

Changes to Callable SM

- You must now include the library utils.a in addition to plotsub.a and devices.a when linking to SM as a graphics library.

- The library names have been changed by pre-pending a 'lib', so you can now say either 'libdevices.a' or '-ldevices'. This allows the -L option to ld to work (under Unix).

- If you compiled SM with an ANSI compiler, you should include the file **"sm_declare.h"** at the top of any C code that calls SM functions.

Bug fixes: (only the ones most visible to users are noted)

- You can now define arbitrarily big vectors with the cursor.

- LOCATION now works with non-touching WINDOWs.