This file consists of Chapter 8 of **MacAnova User's Guide** by Gary W. Oehlert and Christopher Bingham, issued as Technical Report Number 617, School of Statistics, University of Minnesota, revised August 1998, describing Version 4.07 of MacAnova.

Fonts used in this manual are Palatino, Courier, and Symbol.

For information concerning MacAnova, write University of Minnesota, Department of Applied Statistics, 352 Classroom Office Building, 1994 Buford Avenue, St. Paul, MN 55108-6042.

8 Advanced Features

8.1 **MacAnova options** MacAnova's behavior is in part controlled by the value of certain "options." Things affected include the default format for output (Sec. 7.4.1 and 7.4.2), the prompt that is used, seeds used by `rnorm()`, `runi()` and `rpoi()` (Sec. 2.13.1), and whether or not *F*-statistics will be computed and printed in an Analysis of variance (Sec. 3.7, 3.8).

8.1.1 **getoptions()** The current values of all options can be retrieved by `str <- getoptions()` which saves them in a structure whose components have the same names as the options.

```
Cmd> setoptions(default:T) # see Sec. 8.1.3
Cmd> str <- getoptions(); print(str) # Macintosh defaults
component: seeds
(1)            0            0     Values at start up
component: nsig
(1)            5
component: format
(1) "12.5g"
component: wformat
(1) "16.9g"
component: angles
(1) "radians"
component: height
(1)           25
component: width
(1)           80
component: errors
(1)            0
component: prompt
(1) "Cmd> "
component: batchecho
(1) T
component: restoredel
(1) T
component: dumbplot
(1) F
component: scrollback
(1) F
component: missing
(1) "MISSING"
component: warnings
(1) T
component: fstats
(1) F
component: pvals
(1) F
component: fontsize
(1)            9
component: font
(1) "McAOVMonaco"
```

```
component: maxwhile
(1)         1000
component: labelabove
(1) F
component: labelstyle
(1) "("
component: inline
(1) T
component: savehistry
(1) T
component: history
(1)           100
```

See Sec. 8.1.3 for explanations of these options.

You can retrieve the values of specific options by, for example,

```
Cmd> getoptions(format:T,wformat:T)
component: format
(1) "12.5g"
component: wformat
(1) "16.9g"
```

If you name more than one option as was done here, `getoptions()` returns a structure with appropriately named components. Otherwise it returns a scalar or vector.

```
Cmd> str <- getoptions(all:T,format:F,wformat:F)
```

returns the values of all options except `format` and `wformat`.

8.1.2 **setoptions()** You change options using command `setoptions()`. Here is a typical usage:

```
Cmd> setoptions(nsig:7, missing:"NA")
```

This sets the default number of significant digits that are printed to 5 and specifies that a `MISSING` value will be printed as "`NA`" instead of "`MISSING`" .

Each option is set with a keyword phrase of the form `optionName:value`, where `optionName` is the name of the option to be set to `value` which may be `REAL`, `LOGICAL`, or `CHARACTER` depending on the option. See Sec. 8.1.3 for a list of all options.

If `Options` is a structure with component names matching any or all of the legal option names, `setoptions(Options)` sets the options from the component values. Thus

```
Cmd> setoptions(structure(nsig:7, missing:"NA"))
```

does the same as the previous example. This usage also allows you to save all options, change one or more and then restore the original values.

```
Cmd> getoptions(angles:T)
(1) "radians"
Cmd> str <- getoptions() # save all option values
Cmd> setoptions(angles:"degrees")
```

```
Cmd> getoptions(angles:T)
(1) "degrees"          Option angles has changed

Cmd> setoptions(str) # restore all option values

Cmd> getoptions(angles:T)
(1) "radians"          Option angles is as originally
```

You can reset all options to their default values by `setoptions(default:T)`.

8.1.3 **List of available options** Not all of the following are meaningful in all MacAnova versions. They are listed in alphabetical order.

angles, value is CHARACTER scalar `"radians"`, `"degrees"` or `"cycles"`
`setoptions(angles:units)` specifies the angular units assumed for `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, and `atan()` (Sec. 2.8.6), as well as `cpolar()`, `hpolar()`, `crect()`, `hrect()`, and `unwind()` (Sec. 5.2.4). The legal values for `units` are the strings `"radians"` (the default), `"degrees"` (360° equivalent to 2π radians), and `"cycles"` (1 cycle equivalent to 2π radians).

```
Cmd> setoptions(angles:"degrees");vector(acos(.5),cos(150))
(1)          60     -0.86603
```

batchecho, value is `T` or `F`
`setoptions(batchecho:F)` suppresses the normal echoing of the commands read from a batch files (see Sec. 7.6). `setoptions(batchecho:T)` enables such echoing. The value of option `batchecho` is ignored when keyword `echo` is used on `batch()`.

dumbplot, value is `T` or `F`
`setoptions(dumbplot:T)` makes all graphs "dumb" printer plots unless you use keyword phrase `dumb:F` on a plotting commands. See Sec. 8.5.2. The value of option `dumbplot` is ignored when keyword `dumb` is used on a plotting command.

errors, value is non-negative integer
`setoptions(errors:n)` sets the maximum number of errors tolerated to `n`. `n` = 0 means errors will not be counted. See Sec. 8.2 for information on how this option affects how MacAnova handles errors.

format and **wformat**, values are CHARACTER scalars
`setoptions(format:Format)`, sets the default format for printing to the value of `Format`, a quoted string or CHARACTER scalar. For example, after `setoptions(format:"9.4g")` or `setoptions(format:"g9.4")`, most numbers will be printed in floating point form with 4 significant digits and a width of at least 9 characters. `Format` must have one of the forms "*w.d*g" or "*w.d*f" (or "g*w.d*" or "f*w.d*"), where *w* and *d* are integers. For both, *w* is the width, that is, the minimum number of character positions normally used. If *w* is omitted, as in ".8g" or "g.8"), it is computed as *w* = *d*+7.

The specified width is actually only a minimum. If more space is required to provide *d* significant digits or decimals, numbers printed will be wider than *w*.

If the format is "*w.d*g" or "g*w.d*" , output will be in *floating point* form with *d*

significant digits.

```
Cmd> setoptions(format:"14.7g"); vector(1e4*PI,-1e-7*PI)
(1)       31415.93  -3.141593e-07
```

If Format is "*w.d*f" or "f*w.d*", output will be in *fixed point* form with *d* digits after the decimal point; if *d* is zero, no decimal point will be printed.

```
Cmd> setoptions(format:"14.7f"); vector(1e4*PI,-1e-7*PI)
(1)  31415.9265359     -0.0000003
```

The printing of numbers by print() (Sec. 7.4.1) and matprint() (Sec. 7.4.2) as well as many other commands such as anova(), regress(), and cluster(), is controlled by this format.

Setting this option also sets option nsig to *d*.

```
Cmd> setoptions(format:"5.3f"); getoptions(nsig:T)
(1) 3.000      Width 5 with 3 digits after decimal
```

setoptions(wformat:Format) sets the default format just for commands write() and matwrite(). It has no effect on option nsig.

```
Cmd> setoptions(wformat:".19g");write(vector(1e4,-1e-7)*PI)
VECTOR:
(1)        31415.9265358979319  -3.14159265358979321e-07
```

The value options format and wformat are ignored when keywords format or nsig are used on output commands (Sec. 7.4.1, 7.4.2).

font and **fontsize**, values are CHARACTER scalar and positive integer, respectively
setoptions(font:"Courier"), say, changes the font used in the current command window to Courier. In place of "Courier" you can use the name of any available font. Although no check is made, you should always use a non-proportional font (all characters have the same width).

setoptions(font:"Courier 10"), say, changes the font and the font size.

setoptions(fontsize:12), say, changes the font size to 12.

At present, these options are available only in the Macintosh version.

fstats and **pvals**, values are T or F
setoptions(pvals:T) changes the default behavior of many GLM commands to compute and print the *P*-values of various *F*, χ^2 and Student's *t* test statistics under standard assumptions.

setoptions(fstats:T) changes the default behavior of GLM commands producing analysis of variance tables to compute and print the values of *F*-statistics. Unless suppressed by pvals:F on the GLM command, *P*-values will also be printed by default.

height and **width**, values are positive integers
setoptions(height:n) sets the assumed number of lines on the screen to be n. n = 0 means no limit. In *non-windowed* versions of MacAnova, if n > 0, when output from a command fills up the screen, MacAnova will pause and print

```
Hit RETURN to continue or q RETURN to go to next command line:
```

or

```
Press 'q' to quit, 'j' or 'n' to see next line, any other key to continue
```

to keep output from scrolling off the screen. In all versions, option `height` also affects the default maximum number of stems in a stem and leaf display (Sec. 2.12.2) and the size of a "dumb" plot (see Sec. 8.5.2). For compatibility with earlier versions, `lines` is recognized as a synonym for `height`.

`setoptions(width:n)` sets the assumed number of characters on a line to be `n`. The value of `n` must be at least 30. This number, together with the current formatting option, determines how many items are printed per line and the width of "dumb" plots.

On windowed versions, `height` and `width` may be changed when you resize the command window.

The value of options `height` and/or `width` are ignored if keywords `height` and/or `width` are used on plotting or printing commands.

history, value is non-negative integer

`setoptions(history:75)` specifies that the number of previous commands that will be saved for recall and possible editing is 75. This is operative on windowed versions (Macintosh, Windows, Motif) versions, the extended memory version for DOS, and most non-windowed Unix versions. See Sec. 8.8.2, 8.8.3.

inline, value is `T` or `F`

`setoptions(inline:F)` sets the default expansion mode for macros to be out-of-line rather than in-line. `setoptions(inline:T)` restores the usual default. When defining a macro using `macro()`, keyword `inline` takes precedence over this option. See Sec. 9.3.5.

labelabove, value is `T` or `F`

`setoptions(labelabove:F)` specifies that, when non-labeled variables are printed, the labels for the last coordinate (the only coordinate for vectors) are printed across the top, rather than on the left side. It has no effect on the printing of `CHARACTER` variables or of variables with labels (See. Sec. 8.4).

```
Cmd> array(run(16),2,2,4) # default labeling
(1,1,1)           1           5           9          13
(1,2,1)           3           7          11          15
(2,1,1)           2           6          10          14
(2,2,1)           4           8          12          16

Cmd> setoptions(labelabove:T)

Cmd> array(run(16),2,2,4)
                (1)         (2)         (3)         (4)
(1,1)             1           5           9          13
(1,2)             3           7          11          15
(2,1)             2           6          10          14
(2,2)             4           8          12          16
```

labelstyle, value is CHARACTER scalar, one of "(", "[", "{", "<", "/" or "\\"
setoptions(labelstyle:"["), for example, specifies that, when non-labeled variables are printed, coordinate indices are of the form [3], or [3,4]. This operates independently of option labelabove.

```
Cmd> setoptions(labelstyle:"{", labelabove:T)

Cmd> array(run(16),2,2,4)
                {1}          {2}          {3}          {4}
{1,1}             1            5            9           13
{1,2}             3            7           11           15
{2,1}             2            6           10           14
{2,2}             4            8           12           16
```

This option also affects the way a label of the form rep("@",n) is expanded when it is printed. See Sec. 8.4.1.

maxwhile, value is integer ≥ 10
setoptions(maxwhile:2000) specifies that that the maximum allowed repetitions of a while loop is 2000 instead of the default 1000. See Sec. 9.2.3.

missing, value is CHARACTER scalar with no more than 20 characters
setoptions(missing:"NA"), for example, changes the representation of MISSING values (See Sec. 2.7) in output to NA instead of the default MISSING. You can use any CHARACTER scalar instead of "NA".

nsig, value is positive integer
setoptions(nsig:d) specifies that all output except that produced by write() and matwrite() (see Sec. 7.4.1, 7.4.2), should be in floating point format with d significant digits and is equivalent to setoptions(format:*w*.dg), where *w* = d+7. Changing option nsig has no effect on option wformat.

```
Cmd> setoptions(nsig:6);  print(PI)
PI:
(1)       3.14159

Cmd> getoptions("format")
component: format
(1) "13.6g"
```

Option nsig is ignored when formatting information is supplied on print() and matprint().

prompt, value is CHARACTER scalar
setoptions(prompt:newPrompt) changes the prompt from "Cmd> " to newPrompt, where newPrompt is a quoted string or CHARACTER scalar no more than 20 characters long.

```
Cmd> setoptions(prompt:"Next? ")

Next?
```

When setoptions(prompt:Prompt) is executed in a batch file (see Sec. 7.6), the new prompt remains in effect only until the commands in the file are finished. Since a start up file (Sec. 7.8.1) is executed as a batch file, this option cannot be usefully set in a start up file since the prompt is forgotten when the batch file is

completed.

scrollback, value is T or F
setoptions(scrollback:T) changes the default behavior so that when the output generated by a command is so long that its beginning scrolls out of sight, the output window is automatically scrolled back to show the previous prompt after the next prompt is printed. This is available only on windowed versions (Macintosh, Windows, Motif). setoptions(scrollback:F), suppresses such automatic scrolling back. After such a scrolling back, typing anything scrolls the new prompt into view. The value of option scrollback is ignored on help() when keyword scrollback is used.

savehistry, value is T or F
setoptions(savehistry:F) changes the default behavior of save() and asciisave() so that the history of recent command lines will be not be saved. setoptions(savehistry:T) restores the normal behavior so that such a history is automatically saved and will be automatically restored by restore(); when the value is False, the history is not saved. Option savehistry is ignored when keyword history is used on save() and asciisave(). The default value of savehistry is True except in non-interactive mode.

seeds, value is a vector of two positive integers
setoptions(seeds:vector(m,n)) is equivalent to setseeds(m,n), except that setoptions(seeds:vector(0,0)) doesn't initialize the seeds based on the date and time. See Sec. 2.13.1.

warnings, value is T or F
setoptions(warnings:F) suppresses the printing of any lines starting "WARNING:" and setoptions(warnings:T) enables the printing of such lines. This can be useful, for example, if you are doing a lot of arithmetic with variables containing MISSING values which normally generates warning messages. However, setoptions(warnings:F) can be quite dangerous in that many important messages take the form of warnings.

8.2 **Treatment of errors** MacAnova attempts to keep track of the number of errors that occur. What it actually counts is the number of printed messages starting with "ERROR:". If the count reaches a certain threshold, execution is terminated. In interactive mode, the default limit is infinite, so that errors in typing commands should never cause MacAnova to shut down. The default limit is 1 while commands in a batch file are being executed, so that just one error will terminate reading the batch file and return to the prompt level.

You can change the limit for batch files by option errors on the setoptions() command (Sec. 8.1). setoptions(errors:0) or setoptions(errors:1) specifies the default behavior, while setoptions(errors:n) where n is an integer ≥ 2 raises the limit. When setoptions(errors:n) is executed in a batch file, the new value is forgotten when MacAnova returns to the prompt level, but is inherited by any nested batch() commands.

Here is a brief example. Suppose file, mybatch.txt looks like the following with a missing ")"on the first line:

```
delete(indvar  # this is an error: missing ')'
indvar <- run(10) # we've gotten past the error
depvar <- rnorm(10)
regress("depvar=indvar",silent:T)
```

Here is an example of what happens when option errors has value 0 and a larger number.

```
Cmd> setoptions(errors:0) # or setoptions(errors:1)

Cmd> batch("mybatch.txt")

mybatch.txt> delete(indvar  # this is an error: missing ')'
ERROR: missing ')' near delete(indvar
WARNING: too many errors on batch file mybatch.txt

Cmd> # back at the input prompt because of error in batch file

Cmd> setoptions(errors:10) # now allow up to 10 errors

Cmd> batch("mybatch.txt")

mybatch.txt> delete(indvar  # this is an error: missing ')'
ERROR: missing ')' near delete(indvar

mybatch.txt> indvar <- run(10) # we've gotten past the error

mybatch.txt> depvar <- rnorm(10)

mybatch.txt> regress("depvar=indvar",silent:T)

mybatch.txt> (end of file on mybatch.txt)

Cmd> # back at the input prompt because batch file finished
```

8.3 **Creating CHARACTER variables** The simplest way to create a CHARACTER variable is to enter it directly using double quotes.

```
Cmd> labels <- vector("height","weight","age")
```

When all the elements being entered are single words, with no embedded spaces or commas, you can use pre-defined macro enterchars whose arguments should not be quoted and need not be separated by commas. If you do use quotes, they will be treated as part of the word. Successive commas or a trailing comma result in entering null strings ("").

```
Cmd> enterchars(height weight,,"age",)
(1) "height"
(2) "weight"
(3) ""              Because of ,,
(4) "\"age\""       Note quotes are part of value
(5) ""              Because of trailing ,
```

You can also combine CHARACTER, REAL, and LOGICAL data into a CHARACTER scalar or vector using paste() (Sec. 8.3.1, 8.3.2, 8.3.3), create CHARACTER variables consisting of arbitrary characters using putascii() (Sec. 8.3.4), and read CHARACTER data from a file using vecread() and matread() (Sec. 7.2).

8.3.1 **Building custom CHARACTER variables – paste()** paste() allows you to construct complex CHARACTER variables "to order." You can use it to combine quoted strings or CHARACTER variables and the values of REAL and LOGICAL variables into a single CHARACTER variable. The resulting variable can then be printed, perhaps as an error message or as part of a customized table of statistical results, or used to label a graph (Sec. 8.5.1) or the coordinates of a variable (Sec. 8.4).

The basic usage of paste() is

```
Cmd> result <- paste(v1,v2,...) # or print(paste(v1,v2,...))
```

Here v1, v2, ... may be REAL or LOGICAL variables or expressions, quoted strings or CHARACTER variables, or macros. Numerical values are translated to strings of characters such as "3.14159", logical values are translated to "T" or "F", and CHARACTER variables and macros are left as is. All the items in the argument list are "pasted" together to make a single CHARACTER variable. The structure of matrices and arrays is ignored, that is, paste(x) and paste(vector(x)) produce the same string.

```
Cmd> paste("The value of PI is",PI)
(1) "The value of PI is 3.1416"

Cmd> x <- matrix(run(6),2); paste("x is",x)
(1) "x is 1 2 3 4 5 6"   Matrix structure is ignored
```

By default, the arguments are separated by a single space in the output. You can specify a different separator or even several separators or no separator using keyword sep.

```
Cmd> paste(sep:"*",run(7),sep:"=",prod(run(7))) # use 2 separators
(1) "1*2*3*4*5*6*7=5040"

Cmd> paste("M","a","c","A","n","o","v","a",sep:"") # no separator
(1) "MacAnova"
```

The last of these shows that sep:"" indicates *no* separation, and also that a sep keyword phrase that is the last argument is treated as if it were before the first argument.

The default format used for numbers is the same as for print() (see Sec. 7.4.1), except that integers are always formatted as integers and leading and trailing blanks are squeezed out. Missing values are printed as "MISSING" (or the current value of option missing, if different; see Sec. 8.1.3). You can override this default using keyword phrase missing:"?", say, to print missing values as "?".

```
Cmd> x <- vector(1,3,5,?,11);paste(x,missing:"?")
(1) "1 3 5 ? 11"
```

8.3.2 **Formatting paste() output** You can customize the format using keywords format, intwidth, charwidth and justify (but not nsig).

Keyword phrase intwidth:w, where w is a positive integer, specifies that all integer REAL values will be printed using at least w characters, with leading spaces inserted if necessary. Similarly, charwidth:w specifies that all CHARACTER values will be padded on the right with enough spaces to make their width at least w, but will not trim them if they are longer than w.

When you specify a width for CHARACTER values that is wider than a CHARACTER argument requires, the argument is normally padded with blanks on the right – that is, it is left justified. You can modify this behavior by using `justify:"right"` or `justify:"center"` as an argument (`justify:"r"` or `justify:"c"` are also recognized). You can restore the default using `justify:"left"` (or `justify:"l"`).

```
Cmd> print(paste(sep:"","|",charwidth:12,"Source",sep:"|",\
    justify:"r","DF",justify:"c","SS",""))
|Source      |          DF|     SS     |
```

Keyword `format` is used as on `print()` (Sec. 7.4.1, 8.1.3), except that when the format is of the form "`.df`" or "`.dg`" where *d* is an integer (for example, "`.5g`" or "`.4f`"), any leading blanks are trimmed away. If the format is of the form "*w*`.df`" or "*w*`.dg`" (for example, "`12.5g`" or "`7.4f`"), where *w* and *d* are integers, leading blanks are kept. Moreover, if `intwidth` has not been specified, all integers will be padded with blanks on the left to bring the width to *w*.

```
Cmd> paste(format:".10f", "PI =", PI,\
"sqrt(PI) =", format:".5f", sqrt(PI))# 2 formats used
(1) "PI = 3.1415926536 sqrt(PI) = 1.77245"

Cmd> paste("sqrt(PI) =", sqrt(PI), format:"10.5f")
(1) "sqrt(PI) =    1.77245"  Number width is 10 characters

Cmd> dfb <- 5; dfe <- 13; ssb <- 33.245; sse <- 25.039

Cmd> print(paste(charwidth:8,format:"13.6g",intwidth:2,\
"Blocks",dfb,ssb,ssb/dfb,format:"7.3f",(ssb/dfb)/(sse/dfe)))
Blocks    5        33.245         6.649   3.452
```

Lines similar to the last example might be used to compute and print a customized ANOVA table for a randomized block design.

An important use of `paste()` is in creating titles and axis labels for plots (see Sec. 8.5.1). Here is a simple example (see Sec. 9.2.3 for information on the use of `for(...){...}`).

```
Cmd>  powers <- run(-.5,1.5,.5)

Cmd> for(@p,powers){
        plot(X:x, NewY:boxcox(y,@p),\
        title:paste("Plot of boxcox(y,",@p,") vs x",sep:""))
    }
```

This plots five graphs with titles "`Plot of boxcox(y,-0.5) vs x`", "`Plot of boxcox(y,0) vs x`", ..., "`Plot of boxcox(y,1.5) vs x`".

The uses of `paste()` are limited only by your ingenuity. For example, suppose you have 3 variables, `y1`, `y2` and `y3`, and you want to compute regressions of each on independent variables `x1`, `x2`, `x3` and `x4`.

```
Cmd> for(i,run(3)){regress(paste("y",i,"=x1+x2+x3+x4",sep:""))}
```

This produces regression output for the regression models "`y1=x1+x2+x3+x4`", "`y2=x1+x2+x3+x4`" and "`y3=x1+x2+x3+x4`" (Sec. 3.4, 3.8).

Suppose you want to split apart structure `temperatures` in Sec. 2.8.16, with each component going into a separate variable `day_1`, `day_2` and `day_3`..

```
Cmd> for(@i,run(ncomps(temperatures))){
    <<paste("day",@i,sep:"_")>> <- temperatures[@i];;}

Cmd> list("day_*") # See Sec. 2.9.1
day_1           REAL   10
day_2           REAL   5
day_3           REAL   14
```

See Sec. 9.5 on indirect specification of variables by <<...>>.

8.3.3 **Creating CHARACTER vectors using paste()** You can also use paste() to create a CHARACTER *vector* instead of a scalar. If var is a REAL, LOGICAL or CHARACTER variable, paste(var,multiline:T) returns a CHARACTER vector with of length nrows(var), with each element a character representation of a row of var. There can be only one non-keyword argument when you use multiline:T. Keyword format is recognized, but the width of the format is ignored (format:"12.5f" is equivalent to format:".5f"). Keywords charwidth and intwidth are ignored except for printing an advisory message.

```
Cmd> x <- matrix(2*run(8),2); paste(x,multiline:T,format:".1f")
(1) "2.0 6.0 10.0 14.0"
(2) "4.0 8.0 12.0 16.0"
```

You can use keyword sep with multiline:T, but its value must be *single* character. In particular, its value cannot be the null string "".

```
Cmd> paste(x,multiline:T,format:".1f",sep:",")
(1) "2.0,6.0,10.0,14.0"
(2) "4.0,8.0,12.0,16.0"
```

Finally, keyword linesep allows you to combine the lines in a single CHARACTER *scalar*, with each line separated by a character you specify. This is best illustrated by examples.

```
Cmd> paste(x,multiline:T,format:".1f",sep:",",linesep:"/")
(1) "2.0,6.0,10.0,14.0/4.0,8.0,12.0,16.0"

Cmd> paste(x,multiline:T,format:".1f",linesep:"\n")
(1) "2.0 6.0 10.0 14.0
4.0 8.0 12.0 16.0"
```

In the second example "\n" indicates the normal end-of-line character, and each line of x becomes a separate line of the result. The line-separating character is not appended to the last line.

8.3.4 **Creating a CHARACTER variable using putascii()** Normally putascii() just outputs characters to the screen or terminal (Sec. 7.4.3). When keyword phrase keep:T is an additional argument, putascii() returns a CHARACTER scalar containing the characters specified by the codes instead of printing them.

```
Cmd> alphabet <- putascii(run(65,90),run(97,122),keep:T)

Cmd> alphabet
(1) "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
```

```
Cmd> asciicodes <- rep("",127)
Cmd> for(@i,run(127)){asciicodes[@i] <- putascii(@i,keep:T);;}
Cmd> paste(asciicodes[vector(77,97,99,65,110,111,118,97)],sep:"")
(1) "MacAnova"
```

See Sec. 9.2.3 for use of `for` and Sec. 8.3.2 for `paste()`.

8.4 **Coordinate labels** When vectors, matrices and other variables are printed, each row normally starts with a numerical label in parentheses, say `(2,6)`, indicating all the subscripts for the first element in the row. You can use setoptions(`labelabove:T`) to change the behavior so that the labels for the last coordinate go across the screen, above the data, and setoptions(`labelstyle:"["`), say, to use produce default labeling like `[2,6]` instead of `(2,6)`; see Sec. 8.1.3.

If you wish, you may replace these default labels entirely with more informative ones. Specifically, you can add arbitrary labels for the coordinates (rows, columns, ...) of vectors, matrices and arrays, and for the components of structures. A label is a CHARACTER vector of the appropriate length – the dimension of the coordinate or the number of components. If a matrix or array `x` has any labels it must have labels for all dimensions, although a label can be of the form `rep("",n)`. However, a labelled structure can have unlabeled components without labels and an unlabeled structure can have components with labels.

The primary function of coordinate and component labels is to make printed output more informative.

In many cases, when `x` has labels, they are propagated to new variables computed from `x` in the many situations. See Sec. 8.4.3 for details.

8.4.1 **Adding labels to a variable – setlabels()** You attach labels to an existing variable using `setlabels()` or create a variable with labels using `vector()`, `matrix()`, `array()`, `structure()`, `strconcat()`, `matread()` and `read()`.

The general usage for `setlabels()` is

```
setlabels(var, labs)
```

where `var` is an existing variable and `labs` is a CHARACTER scalar or vector or, when `ndims(var)` > 1, a structure whose components are CHARACTER scalars or vectors specifying the labels for the different coordinates.

```
Cmd> x <- matrix(hconcat(run(3,5),run(3,5)^2))
Cmd> x # x has no labels as yet
(1,1)           3           9
(2,1)           4          16
(3,1)           5          25
Cmd> setlabels(x, structure(vector("Case 1","Case 2", "Case 3"),\
    vector("X", "X squared"))) # Add 3 row labels, 2 column labels
```

```
Cmd> x
                  X    X squared
Case 1            3            9
Case 2            4           16
Case 3            5           25
```

Normally, as in this example, the value for labels is a structure with as many components as the variable being labeled has dimensions. When there is only one dimension, the value of labels can be a vector instead of a structure. Except for CHARACTER variables, labels for the last dimension always go across the top, regardless of the value of option labelabove (Sec. 8.1.3).

If x already has labels, they are replaced by setlabels().

It is not an error if the number of label vectors or scalars supplied does not match the number of dimensions although a warning message is printed. Extra labels are ignored and missing ones are assumed to be "@" (see below) and will print as coordinate numbers.

```
Cmd> setlabels(x, vector("Case 1","Case 2", "Case 3"))
WARNING: too few vectors of labels supplied to setlabels(); missing
assumed "@"

Cmd> x # the implied "@" labels label columns with numbers
                 (1)         (2)
Case 1             3           9
Case 2             4          16
Case 3             5          25

Cmd> setlabels(x, structure(vector("Case 1","Case 2", "Case 3"),\
    vector("X", "X squared"),"extra")) # 3 components in structure
WARNING: extra vectors of labels supplied to setlabels() are ignored
```

The warning message can be suppressed by silent:T:

```
Cmd> setlabels(x, structure(vector("Case 1","Case 2", "Case 3"),\
    vector("X", "X squared"),"extra"),silent:T) # no warning msg
```

It is an error if the length of a vector of labels supplied for a dimension is more than 1 but does not match that dimension:

```
Cmd> setlabels(x,labels:structure(vector("Case 1","Case 2"),\
    vector("x1","x2"))) # only 2 labels for dimension 1
ERROR: sizes of labels do not match dimensions on setlabels()
```

In this case, the labels for x are not changed.

setlabels(var,NULL) removes any labels from var. It is not an error if var has no labels.

You can use a single quoted string or CHARACTER *scalar* to generate an entire *vector* of labels for a coordinate as follows. Assume the labels are for coordinate i of variable x.

CHARACTER scalar	Expansion
""	rep("",dim(x)[i])
"@anything"	rep("@anything",dim(x)[i])
"#"	vector("1","2",...)
"["	vector("[1]","[2]",...)
"("	vector("(1)","(2)",...)
"{"	vector("{1}","{2}",...)
"<"	vector("<1>","<2>",...)
"/"	vector("/1/","/2/",...)
"\\"	vector("\\1\\","\\2\\",...)
Anything else, say "base"	vector("base1","base2",...)

In the table, "@anything" stands for any CHARACTER scalar starting with "@", including "@".

```
Cmd> setlabels(x,structure("#","[")); x
          [1]          [2]
1           3            9
2           4           16
3           5           25

Cmd> setlabels(x, structure("(","Column ")); x
        Column 1     Column 2
(1)           3            9
(2)           4           16
(3)           5           25

Cmd> y <- array(run(16),2,2,4) # See Sec. 2.8.15

Cmd> setlabels(y, structure("A","B","C")); y
                C1           C2           C3           C4
A1 B1            1            5            9           13
   B2            3            7           11           15
A2 B1            2            6           10           14
   B2            4            8           12           16
```

A label vector of the form rep("@",n) or rep("@anything",n) as would be expanded from "@" or "@anything" (see table) is treated specially at the time it is used to label output. At that time it is further expanded similarly to the way scalar labels that do not start with "@" are expanded when they are created.

rep("@#", n) prints as 1, 2, … .

rep("@[", n) prints as [1], [2], ..., and rep("@(", n) prints as (1), (2), ..., and similarly with "@{", "@<", "@/",and "@\\". rep("@", n) also prints as (1), (2), … .

rep("@anythingelse", n) prints as anythingelse1, anythingelse2, … .

Moreover, if successive coordinates have the same type of "bracket" label starting with "@" created by, say, labels:structure("@[","[@","["), the printed labels are

combined to form a multi-index label such as, say, `[1,2]`.

```
Cmd> setlabels(x,structure("@(","Column ")); x
        Column 1     Column 2
(1)             3            9
(2)             4           16
(3)             5           25

Cmd> setlabels(y,structure("@(","@(","@(")); y  # or ("@","@","@")
               (1)          (2)          (3)          (4)
(1,1)            1            5            9           13
(1,2)            3            7           11           15
(2,1)            2            6           10           14
(2,2)            4            8           12           16
```

A label vector of the form `rep("",n)` to which the scalar `""` expands effectively deletes any labelling of that coordinate.

```
Cmd> setlabels(x, structure("[","")); x# no column labels
[1]             3            9
[2]             4           16
[3]             5           25
```

Although they sometimes appear the same, there is a difference between the labels generated from, say, `"("` or `"["` and those generated from `"@("` or `"@["`. In the first place, adjacent labels generated from `"("` or `"["` do not combine. Compare the following with the example above where labels were specified by `structure("@(","@(","@(")`.

```
Cmd> setlabels(y,structure("[","[","[")); y
                 [1]          [2]          [3]          [4]
[1] [1]            1            5            9           13
    [2]            3            7           11           15
[2] [1]            2            6           10           14
    [2]            4            8           12           16
```

In addition, although labels “propagate” properly when using subscripts (see Sec. 8.4.4), a vector of the form `rep("@(",n)` or `rep("@[",n)` remains a vector of the same form, except possibly with a different length. When printed, this always produces labels `"(1)"`, `"(2)"`, ... or `"[1]"`, `"[2]"`, ...with no gaps. However, although a vector of labels generated from `"("` or `"["` starts out this way, elements may be skipped when subscripts are used, resulting in gaps in the numerical sequence. Compare the following two examples.

```
Cmd> setlabels(y,structure("A","@[","@[")); y[,2,vector(1,4)]
               [1]          [2]
A1 [1]           3           15
A2 [1]           4           16

Cmd> setlabels(y,structure("A","[","[")); y[,2,vector(1,4)]
               [1]          [4]
A1 [2]           3           15
A2 [2]           4           16
```

You can attach labels to a variable when it is *created* by including keyword phrase `labels:labs` as an extra argument to one of the functions `vector()`, `matrix()`,

array(), structure(), strconcat(), matread() and read(). labs is a CHARACTER scalar or vector or a structure whose components are CHARACTER scalars or vectors exactly as for setlabels(). Here is an alternative way to do the first setlabels() example above:

```
Cmd> x <- matrix(hconcat(run(3,5),run(3,5)^2),\
    labels:structure(vector("Case 1","Case 2", "Case 3"),\
    vector("X", "X squared")))

Cmd> x
                X    X squared
Case 1          3            9
Case 2          4           16
Case 3          5           25
```

Here we attach column labels, but no case labels, to data read from a file:

```
Cmd> iris <- matread("MacAnova.dat","irisdata",\
labels:structure("",\
vector("Variety","Sep_len","Sep_wid","Pet_len","Pet_wid")),\
quiet:T)

Cmd> iris[run(3),] # no row labels
      Variety      Sep_len      Sep_wid      Pet_len      Pet_wid
            1          5.1          3.5          1.4          0.2
            1          4.9            3          1.4          0.2
            1          4.7          3.2          1.3          0.2
```

As with setlabels() it is not an error to provide too many or too few sets of labels. Unlike setlabels(), it is not an error to supply a label vector of the the wrong length. If you do, a warning message is printed, but the operation is carried out ignoring the labels.

```
Cmd> x <- matrix(x,labels:structure(vector("Case 1","Case 2"),\
    vector("x1","x2"))); x# only 2 labels for dimension 1
WARNING: sizes of labels do not match dimensions on matrix(); ignored
(1,1)           3            9
(2,1)           4           16
(3,1)           5           25
```

As with setlabels(), you can suppress warning warning messages by silent:T.

You can remove labels from a variable by setting them to NULL.

```
Cmd> x1 <- array(x,labels:NULL) #works for vector, matrix or array x

Cmd> str1 <- strconcat(str,labels:NULL) # works for structure str
```

For a structure, this does not remove labels from any component with labels.

See Sec. 2.8.10, 2.8.13, and 2.11.3 for other examples of the use of keyword labels.

8.4.2 **Retrieving labels from a variable – getlabels() and haslabels** Function getlabels() allows you to access the labels, if any, of a variable, and pre-defined macro haslabels lets you test whether a variable has labels.

getlabels(x) retrieves the labels, if any, associated with all coordinates of variable x. When x is a vector or structure, the result is a CHARACTER scalar or a CHARACTER vector

of length ncomps(x) or length(x). Otherwise the result is a structure with CHARACTER components named dim1, dim2, ... Each component is either a scalar or a vector of length dim(x)[i]. A scalar consisting of the first label for a coordinate is returned only when all the labels for that coordinate are identical and either are "" or start with "@". Effectively, non-essential elements are trimmed from a vector of labels. When x has no labels, getlabels(x) returns NULL and prints a warning message.

When x is a vector or structure, the result is a CHARACTER vector. Otherwise the result is a structure with CHARACTER vector components named dim1, dim2, ...

getlabels(x,trim:F) does the same, except non-essential elements are not trimmed from a vector of labels that are all "" or all the same and starting with "@".

```
Cmd> temp <- getlabels(iris); list(temp)
temp            STRUC  2

Cmd> compnames(temp)
(1) "dim1"
(2) "dim2"

Cmd> temp
component: dim1
(1) ""     Only 1 label because all labels are ""
component: dim2
(1) "Variety"
(2) "Sep_len"
(3) "Sep_wid"
(4) "Pet_len"
(5) "Pet_wid"

Cmd> temp <- getlabels(iris,trim:F); length(temp$dim1)
(1)          150   All 150 copies of "" returned with trim:F
```

getlabels(x,2 [,trim:F]), for example, retrieves the labels associated with dimension 2 of x. The second argument must be a positive integer or vector of positive integers.

```
Cmd> paste(getlabels(iris,2)) # use paste to pack them in 1 line
(1) "Variety Sep_len Sep_wid Pet_len Pet_wid"
```

haslabels(x) is True if and only if x has labels.

```
Cmd> vector(haslabels(iris),haslabels(matrix(iris,labels:NULL)))
(1) T       F
```

On getlabels() or any command adding labels using keyword labels, you can suppress warning messages by keyword phrase silent:T.

```
Cmd> setlabels(y, NULL) # remove labels from y

Cmd> ylabs <- getlabels(y); list(ylabs)
WARNING: argument to getlabels() has no labels    From getlabels()
ylabs           NULL                              From list()

Cmd> getlabels(y,silent:T) # no warning message printed.
```

8.4.3 **Transforming labels** Because many of the mathematical functions such as `log()` and `cos()` accept CHARACTER arguments, you can sometimes use them to generate appropriate labels for transformed variables. In the following example `iris` is the matrix of iris data used in Sec. 8.4.1 and 8.4.2.

```
Cmd> irislabs <- getlabels(iris)

Cmd> logiris <- matrix(log10(iris[,-1]),\
    labels:structure(irislabs[1],log(irislabs[2][-1])))

Cmd> logiris[run(3),]
 log(Sep_len) log(Sep_wid) log(Pet_len) log(Pet_wid)
      0.70757      0.54407      0.14613     -0.69897
       0.6902      0.47712      0.14613     -0.69897
       0.6721      0.50515      0.11394     -0.69897
```

Elements of a
The use of subscripts to extract the components of `irislabs` is explained in Sec. 2.8.16.

8.4.4 **Propagation of labels** MacAnova tries appropriately to label output or side effect variables created from labelled input variables.

The labels of a portion of a variable selected using subscripts are the appropriate portions of the original labels.

```
Cmd> setlabels(x, structure("Case ",vector("X", "X squared"))); x
                  X    X squared
Case 1            3            9
Case 2            4           16
Case 3            5           25

Cmd> x[1,-1]
          X squared
Case 1            9
```

The result of `cos(x)`, `sqrt(x)`, and other transformations of x listed in Sec. 2.8.6 have the same labels as x.

```
Cmd> sqrt(x)
                  X    X squared
Case 1       1.7321            3
Case 2            2            4
Case 3       2.2361            5
```

`x'` has the same label vectors as `x` but in reverse order

```
Cmd> x'
                Case 1       Case 2       Case 3
X                    3            4            5
X squared            9           16           25
```

`sum(x)`, `min(x)`, and other transformation that operate along the first dimension of `x` have labels for the last `ndims(x) - 1` dimensions matching those of `x`. The first dimension is given `"@"` as a label so that it is printed as `"(1)"`.

```
Cmd> sum(x)
                X    X squared
(1)            12           50
```

+x, -x and !x all have the same labels as x.

Suppose OP is a binary operator such as +, -, *, ==, ..., (Sec. 2.8.3 and 2.8.4), but not a matrix operator such as %*%, %c%, %C%, %/% and %\% (Sec. 2.10.4 and 2.10.5) and x and y are variables with compatible dimensions. Then if x has labels, x OP y often has the labels of the left hand operand x. When x does not have labels, x OP y may have the labels of y. The exceptions have to do with operations combining variables with different sizes (see Sec. 2.10.2). Combination with a scalar preserves labels.

```
Cmd> 3*x
                    X    X squared
Case 1              9           27
Case 2             12           48
Case 3             15           75
```

This can lead to unexpected results. For instance, after regress("y = x1+x2"), side effect variable COEF (See Sec. 3.6) is labelled:

```
Cmd> COEF # elements are labeled with term names
     CONSTANT           x1           x2
          1.3         -5.1          4.1
```

If you use COEF to compute a predicted value, the result will be labelled but the label depends on the order of the terms, since when both operands are labelled, the label of the left operand is used.

```
Cmd> COEF[1] + COEF[2]*7 + COEF[3]*4
     CONSTANT
          -18

Cmd> COEF[3]*4 + COEF[2]*7 + COEF[1]
            x2
          -18
```

In both cases, the label is the label associated with the left most term.

If matrices x and y both have labels then x %*% y, x %c% y, and x %C% y have labels taken from the row and or column labels of x and y in the obvious way.

```
Cmd> x' %*% x
                      X    X squared
X                    50          216
X squared           216          962

Cmd> x %*% x'
              Case 1       Case 2       Case 3
Case 1            90          156          240
Case 2           156          272          420
Case 3           240          420          650
```

When one operand has no labels, the corresponding labels of the product are all "@", yielding numerical labels when printed.

```
Cmd> rep(1,nrows(x))' %*% x # numerical row labels
                X    X squared
(1)            12           50
Cmd> x %*% rep(1,ncols(x)) # numerical column labels
                  (1)
Case 1             12
Case 2             20
Case 3             30
```

When a is a square matrix with labels, the row and column labels `solve(a)` are the column and row labels of a, respectively.

When b is a compatible matrix with labels, the row and column labels of `solve(a,b)` (`a %\% b`) are the column labels of a and b, respectively, and the row and column labels of `rsolve(a,b)` (`b %/% a`) are the row labels of b and a, respectively. If b has no labels, labels of the form `rep("@",m)` are assumed. See 2.10.5

When x is a matrix with labels, `eigen(x)$vectors` and `releigen(x,y)$vectors` (Sec. 6.2.1 and 6.2.3) have the same row labels as x. Similarly the row labels of the matrices of left and right singular vectors computed by `svd()` (Sec. 6.3) are the row and column labels of x, respectively. For all three functions, the column labels of matrices of eigenvectors and singular vectors are of the form `vector("(1)","(2)",...)`, where the parentheses or brackets actually used are determined by option `labelstyle` (Sec. 8.1.3).

When x is a matrix with labels, `cor(x)` (Sec. 2.12.5) has row and column labels matching the column labels of x. `cor(x,y,...)` has no labels.

When x is a matrix with labels, `rft(x)` and `hft(x)` (Sec. 5.10) have the same column labels as x with row labels of the form `rep("@",m)`. The same is true for `cft(x)` when `ncols(x)` is even.

When x is a response variable in a GLM command such as `regress()` or `poisson()`, its labels are propagated to side effect variables `RESIDUALS`, `WTDRESIDUALS`, and `HII` (Sec. 3.6).

After `regress()`, `COEF` and `XTXINV` are labelled with the names of the variables (including `"CONSTANT"` when appropriate) (Sec. 3.6).

After `manova()` (Sec. 3.22) with a response matrix with labels, `SS` is labeled with `TERMNAMES` and two copies of the column labels of the response. Also, the column labels of the response are attached to the last dimension of each vector, matrix, or array returned by `coefs()` and `secoefs()` (Sec. 3.13).

When any term names are longer than 12 characters (the maximum size for a structure component name), the components of `coefs()` and `secoefs()` are labelled with the full term names.

8.5 **More on plotting** A brief introduction to making graphs was given in Sec. 2.15 - 2.15.6. This section gives details on keyword use, modifying and replotting graphs, and saving graphical information on files.

8.5.1 **Keywords affecting appearance and bounds** All the plotting commands recognize but do not require several keyword phrases. Here is a list of keywords affecting graph appearance and bounds, together with brief descriptions. "X-axis" and "Y-axis" refer to the horizontal and vertical axes of a graph.

Key words affecting appearance and bounds	
Keyword Phrase	Explanation
`title:"Your title"`	Title above graph, up to 75 characters
`xlab:"X-axis label"`	X-axis label, up to 50 characters
`ylab:"Y-axis label"`	Y-axis label, up to 20 characters
`xmin:xMinVal`	Minimum value for X-axis
`xmax:xMaxVal`	Maximum value for X-axis
`ymin:yMinVal`	Minimum value for Y-axis
`ymax:yMaxVal`	Maximum value for Y-axis
`xaxis:F`	Do not draw X-axis (line y = 0)
`yaxis:F`	Do not draw Y-axis (line x = 0)
`xticks:xTickPositions`	`REAL` vector of positions of X-axis tick marks. `NULL` means no tick marks or labels; `?` means default positions.
`yticks:yTickPositions`	`REAL` vector of positions of Y-axis tick marks or labels. `NULL` means no tick marks; `?` means default positions.
`xticklen:xTickLength`	Length ≥ –1 of X-axis tick marks; `value` < 0 gives ticks outside frame; 0 gives tick labels but no ticks; value > 2 gives full grid lines across plot; 1 gives the default length.
`yticklen:yTickLength`	Length ≥ –1 of Y-axis tick marks; `value` < 0 gives ticks outside frame; 0 gives tick labels but no ticks; value > 2 gives full grid lines across plot; 1 gives the default length.
`impulse:T`	Draw vertical lines from points to *y* = 0 line
`lines:T`	Connect points with straight lines
`linetype:n`	On commands that draw lines, sets the line type to `n`, default is 1 (solid); `n` must be integer 1 ≤ `n` < 100.
`thickness:w`	On commands that draw lines, sets the line thickness to `w` times normal thickness, default is 1. `w` must be between .1 and 10; has no effect when when `dumb:T` or where otherwise not feasible; not implemented in all versions.

If the values for `xmin` and `xmax` and/or `ymin` and `ymax` are the same (for example, `xmin:0,xmax:0`), bounds for the X and/or Y axis are computed from all the data in the plot.

8.5.2 **Other graphics keywords** These keywords allow saving graphs in files, directing them to specific windows, and adding information to previously created plots. Here is a list of the remaining graphics keywords.

Other Graphics Keywords	
Keyword Phrase	Explanation
`dumb:T`	Use printable characters only to produce a low resolution plot suitable for typewriter-like printing
`height:nlines`	Number of lines in a "dumb" plot
`width:nchars`	Width of a "dumb" plot in character positions.
`keep:F`	Do not save plot as `LASTPLOT` (see Sec. 8.5.3)
`show:F`	Do not display plot, only save it as `LASTPLOT` (see Sec. 8.5.3)
`add:T`	Add information to most recent plot
`file:fileName`	Write Postscript to file `fileName` (see Sec. 8.5.4)
`new:T`	Overwrite `fileName` (see Sec. 8.5.4)
`ps:F`	Suppresses PostScript when writing a plot to a file (see Sec. 8.5.4). On Unix this results in the Tektronix plotting commands being written to the file; on a Macintosh, a PICT file is written; on other computers, a "dumb" plot is written.
`epsf:T`	Encapsulated PostScript file will be written instead of PostScript (Macintosh only).
`landscape:T`	PostScript plot will be rotated so as to fill 8.5" by 11" page.
`window:n`	Draw plot in window `n` ($1 \le n \le 8$). If `n` is 0, use the window most recently used; only on windowed versions (Macintosh, Windows or Motif).
`pause:T` (Mac,Window, Motif) `pause:F` (DOS,Unix)	Forces (`T`) or suppresses (`F`) a pause after the graph is drawn. `pause:T` is when plotting many graphs in a loop with `window:0`.
`screendump:FileName`	Save a copy of graph being plotted in file FileName (see Sec. 8.5.4). In the Macintosh version a PICT file is written; in the extended memory DOS version, a bit map PCX file is written. Not legal in other versions.
`notes:Notes`	Attach `CHARACTER` vector or scalar `Notes` to `LASTPLOT` (see Sec. 8.5, 8.9).

Most are self explanatory; `keep` and `show` are explained in Sec. 8.5.3, `file` and `new` are explained in Sec. 8.5.4 and `notes` is explained in Sec. 8.9.

Here are some examples of the use of tick mark related keywords.

```
Cmd> plot(x,y,xticks:vector(1,2,4),yticks:NULL,xticklen:1.5)
```

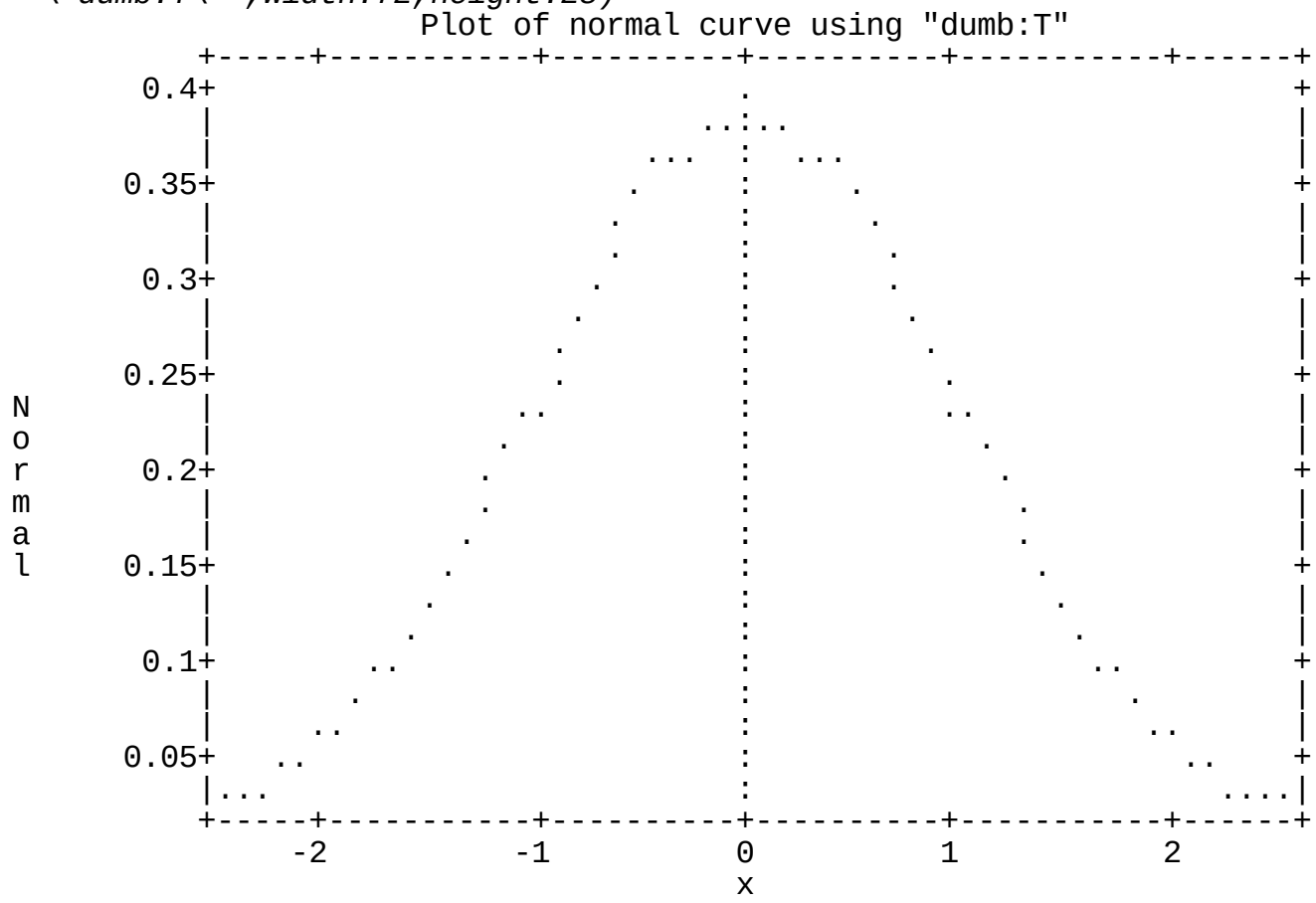
gives X-axis ticks 1.5 times normal at x = 1, 2 and 4 and suppresses all y-axis ticks and their labels.

```
Cmd> plot(x,y,xticklen:3,yticklen:-.5)
```

draws full grid lines (value for `xticklen` > 2) perpendicular to the x-axis and half length ticks along the outside of left edge of the frame.

Here is an example of a plot produced using `dumb:T`.

```
Cmd> @x<-run(-2.5,2.5,.1); lineplot(@x,\
Normal:exp(-@x^2/2)/sqrt(2*PI),\
dumb:T,title:"Plot of normal curve using
\"dumb:T\"",width:72,height:28)
```



When `xticklen` or `yticklen` is used in making a "dumb" plot, the only values that have an effect are 0 (tick marks but not labels are suppressed) and > 2 (grid lines are drawn). For all other values the tick marks are as just illustrated.

8.5.3 **Replotting graphs and GRAPH variables** As a side effect, all plotting commands create a variable with name `LASTPLOT` of special type `GRAPH`. `LASTPLOT` encapsulates all the information used to create the plot. This information includes axis labels and title, minima, maxima, and indeed everything set by keywords affecting appearance and bounds when the plot was created (Sec. 8.5.1). `LASTPLOT` can be assigned to another variable (for example, `graph1 <- LASTPLOT`) or redisplayed, possibly with changed

limits or labelling information, using showplot(). You can add information to it using addpoints(), addlines(), addchars(), and addstrings() or keyword phrase add:T on a regular plotting command. You can print LASTPLOT (as a "dumb" plot) using print() or write(). In fact, just typing the name of a GRAPH variable causes a "dumb" rendition to be printed.

Command showplot() recognizes all the keywords in Sec. 8.5.1 and 8.5.2 except impulse, lines, linetype, thickness and add, and updates LASTPLOT accordingly (unless keep:F is an argument), thus allowing labelling information and plotting limits to be changed.

Here are descriptions of the commands that may be used to add information to a plot in LASTPLOT or another GRAPH variable. If the GRAPH variable does not exist it is an error.

addpoints(x,y) is the same as plot(x,y,add:T). It redraws the graph in LASTPLOT while adding new points to it.

addchars(x,y,c) is the same as chplot(x,y,c,add:T). It redraws the graph in LASTPLOT while adding character labelled points to it.

addlines(x,y) is the same as lineplot(x,y,add:T). It redraws the graph in LASTPLOT while adding line plots to it. addlines(x,y,lines:F) is the same as addpoints(x,y).

addstrings(x,y,charVec) draws charVec[i] at position (x[i],y[i]) in the graph in LASTPLOT. charVec must be a CHARACTER vector of the same length as x and y. In contrast with other plotting commands, both x and y must be vectors of the same length. By default, each string is written *centered* at (x[i],y[i]) . However, if justify:"left" or justify:"right" is an argument following charVec, each string will be positioned with its left or right or right end at (x[i],y[i]).

On windowed versions of MacAnova (Macintosh, Windows, Motif), these commands automatically redraw the most recently drawn window, unless keyword window is used to specify another window (Sec. 8.5.2).

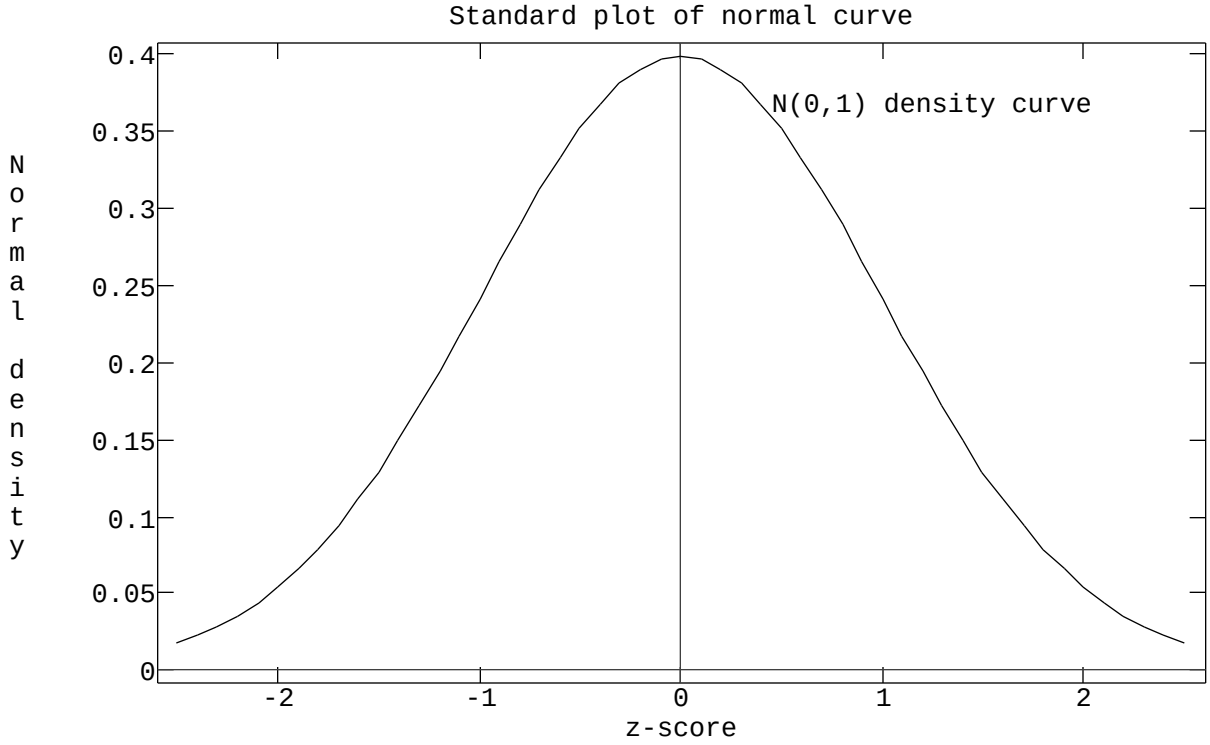
On any of these, to force the minimum or maximum on an axis to be recomputed, use, say, xmin:?,ymin:?. This computation takes into account the minimum and maximum of all previous data as well any new data being added.

All four commands recognize a GRAPH variable as argument preceding any others, for example, addpoints(graph,x,y). The new information or labels will be added to the plot encapsulated in graph instead of LASTPLOT. In addition, plot(graph,x,y), chplot(graph,x,y,c), and lineplot(graph,x,y) act identically to addpoints(graph,x,y), addchars(graph,x,y,c), and addlines(graph,x,y), respectively. You don't need keyword phrase add:T.

As with plot(), chplot() and lineplot(), you can replace arguments x and y by a structure whose first two components are interpreted as x and y.

The following assumes that LASTPLOT was created by the dumb:T example above. Note the use of keywords to change the axis labels and titles.

```
Cmd> normal <- LASTPLOT # save a copy of the GRAPH variable
Cmd> addstrings(normal,.45,.37,"N(0,1) density curve",\
    justify:"left",ymin:0,xlab:"z-score",ylab:"Normal density",\
    title:"Standard plot of normal curve")
```



GRAPH variable normal has not been changed, but LASTPLOT now encapsulates all the components of this graph.

All plotting commands, including showplot(), addpoints(), addlines(), addchars(), addstrings() and boxplot(), recognize keyword phrases keep:F and show:F. keep:F specifies that LASTPLOT is *not* to be created or updated and show:F directs that the graph is not displayed. The latter is useful when you are building a graph in several steps and don't want to see anything until the final product. For example, the preceding plot could be created by

```
Cmd> @x<-run(-2.5,2.5,.1);lineplot(@x,\
    exp(-@x^2/2)/sqrt(2*PI),show:F)
Cmd> addstrings(.45,.37,"N(0,1) density curve",\
    justify:"left",show:F)
Cmd> showplot(ymin:0,xlab:"z-score",ylab:"Normal density",\
    title:"Standard plot of normal curve")
```

Using both keep:F and show:F doesn't make sense and is an error.

8.5.4 **Writing graphs to a file** All plotting commands recognize keyword phrase file:fileName, where fileName is a quoted string or CHARACTER variable. This suppresses the display of the graph. Instead, plotting information is written as PostScript commands to the file specified by fileName. (PostScript is a page description

language that is recognized by many printers, include Apple LaserWriters.) It may be possible to the print the PostScript commands later on a LaserWriter or incorporate them in a document. You may write several graphs to the same file. You should use `new:T` when saving the first one. Without `new:T`, if the file already exists, Postscript commands are added at its end. On windowed versions (Macintosh, Windows, Motif) you can use `""` as file name, specifying the file in a dialog box. File name `CONSOLE` is not treated specially by the plotting commands.

Note: If you add PostScript to a file that was written in a previous MacAnova run, some programs designed to read the PostScript may be unable to read the new plots. In particular this is true of Unix program `ghostview`.

Actually, you can suppress the writing of Postscript by keyword phrase `ps:F`. What is written in this case depends on the MacAnova version. On Unix, high resolution graphs are available only if you are using a Tektronix 4014 emulator (the `xterm` interface under X-windows is one such) and MacAnova emits special Tektronix 4014 character sequences. `ps:F` causes the Tektronix 4014 codes to be written to the file instead of Postscript. When `ps:F` is used on a Macintosh a so called `PICT` file is written that can be read by many graphics programs. On other systems, when `ps:F` is present, just a "dumb" plot is written to the file.

In versions where it is implemented (Macintosh and protected mode DOS), keyword phrase `screendump:fileName` provides an alternative way to save a graph. A file is written to the file in a binary format specific to the type of computer. On a Macintosh a PICT file is written; in the protected mode DOS version a bit map PCX file is written. Such files can be imported into certain word processors and graphics editing programs.

8.6 **More on help() and usage()** `help()` and `usage()` have additional features not mentioned in Sec. 2.9. You can switch between two different help files and find out what's new in the version of MacAnova you are using.

8.6.1 **Using more than one help file** `help()` and `usage()` read information in a special format from a text (ASCII) file. On non-Unix systems this file is `MacAnova.hlp` and is normally in the same directory or folder where MacAnova itself is located . On Unix, it will be in an installation-dependent location and may have an alternative name. The format is described near the start of `MacAnova.hlp`.

It is possible to create additional files in the same format, perhaps providing help on a library of macros (see Sec. 7.5), or specific help on one or more statistical methods. For example, files `design.mac` and `tser.mac`, which are distributed with MacAnova, contain macros useful in experimental design and time series, respectively. Help files `design.hlp` and `tser.hlp`, also distributed with MacAnova, provide help for these macros.

You can use `keywords` `file`, `orig` and `alt` on `help()` to switch between help files. This is probably best illustrated by example.

```
Cmd> help(file:"design.hlp") # start using design.hlp as help file

Cmd> help(confound3) # help on topic will be read from design.hlp
confound3(basis) confounds a three series factorial into blocks based
on the generators given in the matrix basis.  Results are returned in
a structure with component names 'block1', 'block2', etc.  Each
component has a CHARACTER vector of factor/level combinations for
that block.

The p x k matrix basis contains the generators for the confounding,

*****  Interrupt  *****  Interrupted to stop output

Cmd> usage(randt) # usage information from design.hlp
randt(dvec, m [,trials:n]), REAL vector dvec, positive integer n

Cmd> help("*") # all topics on design.hlp
Help is available on the following topics:
aliases2     allaliases2  confound3    mixed        randt
aliases3     boxcoxvec    design_index pairedcomp   rscanon
all3anova    choosegen2   ems          quadmax      typeIIIss
all4anova    confound2    ffdesign2    randsign     varcomp
For help on topic foo, enter help(foo) or help("foo")

Cmd> help(key:"?") # index keys for design.hlp
Type 'help(key:"heading")', where heading is in following list:
Aliasing         ANOVA            Design           Permutation test
Analysis         Confounding      Factorial        Random effects
```

`design.hlp` will remain the source for `help()` and `usage()` until you change it.

`help(orig:T)` switches you back to the standard help file; `help(alt:T)` returns you the most recent alternative help file.

```
Cmd> usage(orig:T) # or help(orig:T)

Cmd> usage(boxcox) # usage info now from macanova.hlp
boxcox(x,power), x a REAL vector or matrix, power a REAL scalar

Cmd> help(alt:T) # switch back to most recent alternate help file

Cmd> usage(boxcoxvec) # usage info again from design.hlp
boxcoxvec(rhs_model,y,powers:pow), CHARACTER scalar rhs_model, REAL
  vectors y and pow.
```

You can combine other possible arguments with these keywords. For example, `help(alt:T,key:"factorial")` both switches to the alternate file and lists the topics associated with index key `factorial` in that file.

8.6.2 **Finding what's new** MacAnova is an evolving system and is not likely to remain unchanged for long. Changes may be bug fixes, the addition of new functions or macros, or the enhancement of existing ones. Whenever a substantive change is made, including important bug fixes, a record is made in the help file under the general topic `news`, along with the date the change was made. Especially when you start using a new version, you will probably want a quick synopsis as to what has changed.

`help(news)` lists in reverse chronological order items about MacAnova development starting with the most recent entry back for three months from the most recent date.

`help(news:971001)` gives you information about changes since October 1, 1997.

`help(news:vector(970101,970630))` lists in reverse chronological order items dated between January 1 and June 30, 1997.

```
Cmd> help(news:vector(980725,980731)) # between July 25 and 31, 1998
980731 keyvalue() argument specifying the keyword name can contain
the "wild card" character '*' so that, for example, keywords 'pow',
'power', and 'powers' will all match "pow*".

980727 New functions setlabels(), attachnotes() and appendnotes()
allow attaching labels and notes to existing variables.  New pre-
defined macro hasnotes tests whether a variable has notes.
```

From time to time, older `news` items are moved from the standard help file to file `macanova.nws` which is distributed with MacAnova. This is in the form of an alternate help file. `help(file:"macanova.nws",news)` will list the most recently added items to this file. You can again specify dates to select ranges of items.

`help(update)` prints a summary of most changes between the various versions of MacAnova, in reverse chronological order, going back to Version 2.0.

8.7 **Running other programs from within MacAnova** In the windowed versions of MacAnova (Macintosh, Windows and Motif) you can switch to a non-MacAnova window and start up one or more programs running in parallel with MacAnova. If such a program is a word processor or editor you can transfer text and graphical output to the program by using Copy and Paste on the Edit menu.

When running a non-windowed version in a windowed environment (for example, a DOS version in a Windows 95 DOS window), you can start up parallel programs in the other windows and may be able cut and paste from or to another program's window.

In addition to this capability, in some versions you can run other programs directly from MacAnova. The most versatile method uses command `shell()`. Somewhat simpler to use but less versatile is a "shell escape", a command line whose first character is "!."

Neither `shell()` nor shell escapes are available on the Macintosh.

8.7.1 **shell()** In its simplest form, `shell(cmd)`, runs the system (Unix, DOS/Windows, VMS) command or program in `cmd`, a quoted string or `CHARACTER` variable. Here is a simple Unix or Unix Motif example:

```
Cmd> shell("ls userfun") # list directory userfun
Userfun.h
dynload.h
foo.c
fooeval.c
foosymh.c
goo.c
```

If the operating system allows it, `cmd` can contain more than one command to be run. Here is a Unix example:

```
Cmd> shell("echo line 1;echo line 2") # or use '\n' instead of ';'
line 1
line 2
```

You cannot execute multiple commands in this way on DOS or Windows.

At present (August 1998), shell() does not work predictably in the Windows version.

8.7.2 **shell() keyword phrases interact:T and keep:T** The simplest usage doesn't work right if the user needs to interact with the program being run as would be necessary with an editor. Keyword phrase interact:T enables such interaction. For example, on Unix,

```
Cmd> shell("vi mymacro.mac",interact:T)
```

would allow you to edit file mymacro.mac using program vi. Without interact:T, what you want to edit may not be displayed. In Motif, interaction normally takes place in the window from which you launched MacAnova, *not* the MacAnova window itself.

Depending on the system, with a non-interactive command you may get output formatted slightly differently with interact:T than without it. Compare the following with the example in Sec. 8.7.1:

```
Cmd> shell("ls userfun",interact:T)
Userfun.h  dynload.h  foo.c      fooeval.c  foosymh.c  goo.c
```

Sometimes you would like to be able to save the output of a system command or program in a MacAnova variables. This is possible using keyword phrase keep:T as a second argument to shell(). It this usage, no output is printed, but what would have been printed is returned as a CHARACTER vector, with each line of output, including blank lines, in an element of the vector.

```
Cmd> shell("ls userfun",keep:T)
(1) "Userfun.h"
(2) "dynload.h"
(3) "foo.c"
(4) "fooeval.c"
(5) "foosymh.c"
(6) "goo.c"
```

No interaction with the command or program being run is possible when you use keep:T.

Use of keep:T in the limited memory DOS version of MacAnova is an error.

8.7.3 **Lines starting with "!"** An alternative way to execute a system command or program interactively is with the shell "escape" character "!" in the first character position after the prompt.

```
Cmd> !ls userfun
Userfun.h  dynload.h  foo.c      fooeval.c  foosymh.c  goo.c
```

This is exactly equivalent to shell("ls userfun", interact:T).

Because of this feature, if you want to start a MacAnova command with "!", you must precede it with a space. This is true in all versions, not just those with an operative `shell()` command.

8.8 **Recalling previous commands** All versions of MacAnova save the most recent command line as a macro named `LASTLINE`. In addition, most versions (the limited memory DOS version is one exception) maintain a "history," an internal list of the most recent command lines. If *N* is the value of option `history` (see Sec. 8.1.3), up to *N* commands are remembered. They can be retrieved using function `gethistory()`, by pressing certain key combinations, or, in windowed versions (Macintosh, Windows, Motif) by selecting a menu item. You can replace the internal list using `sethistory()`. In addition, by default, `save()` saves the current history and `restore()` replaces it.

8.8.1 **LASTLINE and macros redo and REDO** Just before executing a command line, MacAnova creates a macro `LASTLINE` whose text is the command line.

```
Cmd> PI*run(4) # some command
(1)       3.1416       6.2832       9.4248       12.566

Cmd> LASTLINE
(1) "PI*run(4) # some command"

Cmd> LASTLINE # now has a new value
(1) "LASTLINE"

Cmd> run(3)
(1)            1            2            3

Cmd> LASTLINE() # previous line reexecuted
(1)            1            2            3
```

As this last command shows, this allows you to re-run the previous command line without re-typing it. You can do so only once, since, for example, after the last command, the value of `LASTLINE` is `"LASTLINE()"`. If you try to run this, `LASTLINE` recursively tries to execute itself, leading to the following error message:

```
ERROR: Parser stack overflow; probably too deep macro recursion
```

Pre-defined macro `redo` takes advantage of this feature to make it easier to repeat the previous command. All `redo` does is to create macro `REDO` by `REDO <- LASTLINE` and then to execute `REDO`. You can now use `REDO` one or more times to execute it yet again.

```
Cmd> sqrt(2)+PI
(1)       4.5558

Cmd> redo()
(1)       4.5558

Cmd> REDO() # redoes the same thing
(1)       4.5558
```

Using `redo` (but not `REDO`) on two command lines in a row is an error, since it then tries to run itself.

8.8.2 **Keyboard and menu recall** In addition to LASTLINE, most MacAnova versions maintain a "history," an internal list of recent command lines that can be recalled to the current command line using certain key combinations or menu items. Here is a table of the permissible key combinations on the various versions.

Version	Up History Keys	Down History Keys
Macintosh	Option+↑ or F7	Option+↓ or F8
Windows	Ctrl+↑ or F7	Ctrl+↓ or F8
DOS extended memory	↑ or Ctrl+P	↓ or Ctrl+N
DOS limited memory	Not available	Not available
Motif	Ctrl +Keypad ↑ or F7	Ctrl +Keypad ↓ or F8
Unix (most versions)	↑ or Ctrl+P	↓ or Ctrl+N

You move backward through the list using the Up History Keys and forward using the Down History Keys. In all the windowed versions you can also select items **Up History** and **Down History** on the **Edit** menu instead of using key combinations. As you successively move back, previous commands appear after the prompt. These can be executed "as is" or edited in place and then executed.

The number of lines you can go back is controlled by the value of option `history` with default value 100. See Sec. 8.1.3.

8.8.3 **gethistory() and sethistory()** On versions that maintain a history of recent commands, `gethistory(n)` returns the `n` most recent commands and `gethistory()` returns all available commands. Here is output obtained just after launching MacAnova, so there is no history available.

```
Cmd> gethistory() # first command after launching
(1) ""     Returns "" when there is no history available

Cmd> 1+1 # a command
(1)            2

Cmd> 1+2 # another command
(1)            3

Cmd> gethistory()
(1) "gethistory() # first command after launching"
(2) "1+1 # a command"
(3) "1+2 # another command"

Cmd> gethistory(2) # get 2 most recent commands
(1) "1+2 # another command"
(2) "gethistory()"
```

One use is as a "scripting" device to create a macro from several previously typed commands. Suppose we want to compute a simple macro that will compute the regression coefficients and residual sum of squares from a cubic polynomial regression of `y` on `x`.

```
Cmd> x <- run(7);y <- vector(-0.23,-2.20,-0.37,-1.41,0.49,0.42,0.28)
```

```
Cmd> x2 <- x*x; x3 <- x*x2

Cmd> regress("y=x+x2+x3",silent:T);vector(COEF,SS[4])
(1)       2.2386      -3.6367       1.0558      -0.0825       1.4701

Cmd> doit <- macro(paste(gethistory(2),multiline:T,linesep:"\n"))

Cmd> doit # here's the macro that was created
(1) "x2 <- x*x; x3 <- x*x2
regress(\"y=x+x2+x3\",silent:T);vector(COEF,SS[4])"

Cmd> # type in new data

Cmd> x <- vector(1, 3, 4, 7, 8); y <- vector(3.2,4.1,7.6,1.5,2.0)

Cmd> doit() # do the regression with the new data
(1)     -0.75573       4.5773     -0.94795     0.051388      0.79738
```

See Sec. 8.3.3 for the use of paste() and Sec. 9.3.1 for the use of macro().

If commands is a CHARACTER vector with elements that are or could be MacAnova command lines, then sethistory(commands) replaces the history list with the elements of commands.

```
Cmd> commands <- vector("z <- 3+4","print(x[run(5)])","\"Hello\"")

Cmd> sethistory(commands)

Cmd> gethistory() # this retrieves the history just set
(1) "z <- 3+4"
(2) "print(x[run(5)])"
(3) "\"Hello\""
```

A complete save() normally saves the current command history, and restore() restores it. See Sec. 7.7 for information on save() and restore() keyword history and option savehistry which can modify this default behavior. Because the command history is saved, if you type gethistory() immediately after restoring a complete workspace, you can see a record of just what you were doing before the workspace was saved. And you can use the arrow keys to re-execute these commands, possibly after editing.

8.9 **"Notes" attached to variables** You can attach CHARACTER vectors as descriptive "notes" to almost any variable, including GRAPH variables and macros. Such notes might describe the origin of the variable or graph or give information on macro usage.

You attach notes to a variable using function attachnotes() or by including notes:CharVec as an argument to vector(), matrix(), array(), structure() or any of the plotting commands (Sec. 8.5.2). You can add additional notes to a variable using appendnotes(). You can retrieve notes from a variable using getnotes() and can test whether a variables has descriptive notes using pre-defined macro hasnotes.

Since all operations and functions except for a getnotes(), matprint(), matwrite(), macrowrite(), save() and asciisave() completely ignore notes, having notes attached to to a variable has no effect on any operations involving it.

matprint(), matwrite() and macrowrite() automatically write any attached notes

in a form that is readable by read(), matread() and macroread(). See Sec. 7.1, 7.4.1, 7.5.1, 7.5.2.

8.9.1 **attachnotes(), appendnotes(), getnotes() and hasnotes** When x is an existing variable of any type except NULL, and Notes is a CHARACTER vector, attachnotes(x,Notes) "attaches" Notes to x. Normally Notes will document what x is. If x is a GRAPH variable (Sec. 8.5.3), Notes might describe the variables plotted, or give information as to how they were computed. When x is a macro, Notes might be information on its usage. If x has been read from a file by read(), matread() or macroread(), a possible source for Notes might be the comment information retrieved by inforead() (Sec. 2.11.5). attachnotes(x,NULL) removes any notes attached to x.

appendnotes(x,Notes) appends additional notes to a variable x. If x has no attached notes, appendnotes(x,Notes) is the same as attachnotes(x,Notes). appendnotes(x,NULL) does nothing.

If x has attached notes, getnotes(x) retrieves them as a CHARACTER scalar or vector. If x has no such notes, getnotes(x) returns NULL and gives a warning message. getnotes(x,silent:T) does the same except that the warning message is suppressed when x has not notes.

```
Cmd> iris <- matread("macanova.dat","irisdata",quiet:T)

Cmd> # attach comment lines from data set in file as notes

Cmd> attachnotes(y, inforead("macanova.dat","irisdata",quiet:T))

Cmd> getnotes(y) # let's see them
(1) " Data from R. A. Fisher, The use of multiple measurements in
 taxonomic problems, Annals of Eugenics 7 (1936) 376-386
 Col. 1: Variety number (1 = I. Setosa, 2 = I. Versicolor,
         3 = I.Virginica)
 Col. 2: X1 = Sepal length
 Col. 3: X2 = Sepal width
 Col. 4: X3 = Petal length
 Col. 5: X4 = Petal width
 Rows 1-50:    Group 1 = Iris Setosa
 Rows 51-100:  Group 2 = Iris Versicolor
 Rows 101-150: Group 3 = Iris Virginica"

Cmd> varieties <- factor(iris[,1])

Cmd> irisdepv <- matrix(iris[,-1],notes:getnotes(y))

Cmd> appendnotes(irisdepv,\
"Variety number has been removed; Col. 1 is now Sepal Length")

Cmd> getnotes(irisdepv)[2] # element 2 of notes
(1) "Variety number has been removed; Col. 1 is now Sepal Length"
```

Another way to attach notes is by including keyword phrase notes:Notes as an argument to vector(), matrix(), array(), structure(), macro() or any of the plotting commands, where Notes is a CHARACTER scalar or vector. Here are a couple of examples.

```
Cmd> rainfall <- vector(21.5,21.1,19.9,19.7,18.4,16.1,\
26.6,16.8,14.2,23.3, notes:"1937 - 1946 Rainfall")

Cmd> plot(year:1937, rainfall, show:F,\
    notes:"Plot of rainfall vs year") # don't display

Cmd> getnotes(rainfall)
(1) "1937 - 1946 Rainfall"

Cmd> getnotes(LASTPLOT)
(1) "Plot of rainfall vs year"
```

The notes in `plot()` are attached to `GRAPH` variable `LASTPLOT`.
Pre-defined macro `hasnotes` allows you to test whether a variables has notes attached.

```
Cmd> vector(hasnotes(iris),hasnotes(PI),hasnotes(LASTPLOT))
(1) T       F       T
```