This file consists of the second part of Chapter 2 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 chapter 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.

2.11.1 **vecread()** Function vecread() creates a REAL vector from unstructured numerical data in a file. The numbers should be separated by blanks or tab characters, or may be on separate lines. One or more successive question marks (?, ??, ???, ...) in the file is read as MISSING as are isolated periods (.) and asterisks (*). A exclamation point "!" is a "stop character" and terminates the read. Any other non-numeric characters such as letters, commas or slashes are ignored except possibly for printing an advisory message. The entire file is scanned as far as the first "!", if any. Here is a listing of a file myfile.dat containing 30 comma-separated values of a variable x that might be read by vecread():

```
Data from trees
600, 555, 361, 489, 640,  ??  644, 297, 481, 612, 522
246, 504, 358, 623, 614, 595, 531, 602, 684, 410, 448
662, 892, 644,   ., 431,   *, 513, 603f ! 30, 10
10, 20, 3 This line will not be read
```

The output of vecread() is a REAL vector consisting of the values read.

```
Cmd> x <- vecread("myfile.dat");x # read from file myfile.dat
WARNING: nonnumeric character(s) ignored on myfile.dat
 (1)         600         555         361         489         640
 (6)     MISSING         644         297         481         612
(11)         522         246         504         358         623
(16)         614         595         531         602         684
(21)         410         448         662         892         644
(26)     MISSING         431     MISSING         513         603
```

Note that ??, . and * were read as MISSING and nothing after the "!" was read. The WARNING message results from the non-numeric first line.

You can specify a different stop character instead of "!" by vecread(filename, stop:"%"), say. The stop character must be a punctuation character other than "+", "-", ",", "?" or ".", that is, any of !"#$%&'()*/:;<=>@[\]^_`{|}~. The stop character can also be a "non-ascii" character specified in octal form as "\200", "\201", "\202", ..., or "\377". This might be called for when you want to ensure the whole file is read, and you know the file does not contain the stop character.

If the file contains lines, all beginning with the same character, say "#", that should be ignored, you can use keyword phrase skip:"#" as an argument. Thus, suppose myfile1.dat looks like

```
# Data on 30 trees
600, 555, 361, 489, 640,  ??  644, 297, 481, 612, 522
246, 504, 358, 623, 614, 595, 531, 602, 684, 410, 448
662, 892, 644,   ., 431,   *, 513, 603f % 30, 10
10, 20, 3 This line will not be read
```

Then vecread("myfile1.dat",stop:"%",skip:"#") will return exactly the same vector as before. If the first line were not skipped, 30 would be read as an item of data.

Another way to control how much of the file is read is by specifying a "go character" with keyword go. vecread() will stop reading on the first line that does *not* start with the go character or the skipping character, if any. Suppose myfile2.dat looks like

```
# Data on 30 trees
 600, 555, 361, 489, 640,  ??  644, 297, 481, 612, 522
 246, 504, 358, 623, 614, 595, 531, 602, 684, 410, 448
 662, 892, 644,   ., 431,   *, 513, 603
Data file created 980501
```

Then `vecread("myfile2.dat",go:" ",skip:"#")`, will return the same data as before, the last line not being read because it starts with `D` rather than a space. If a space were inserted before "`Data file`", then `980501` would be read as an additional data item. You can't specify both a go character and a stop character.

Suppose `myfile3.dat` contains data arranged in columns like the following

```
# Data on 5 variables, n = 6
  600   555   361   489   640
    ?   644   297   481   612
  522   246   504   358   623
  614   595   531   602   684
  410   448   662   892   644
    ?   431     ?   513   603
```

You can read it into matrix `y` as follows:

```
Cmd> y <- matrix(vecread("myfile3.dat",skip:"#",quiet:F),5)'
# Data on 5 variables, n = 6   Echoed because of quiet:F

Cmd> y
(1,1)         600          555          361          489          640
(2,1)     MISSING          644          297          481          612
(3,1)         522          246          504          358          623
(4,1)         614          595          531          602          684
(5,1)         410          448          662          892          644
(6,1)     MISSING          431      MISSING          513          603
```

The use of keyword phrase `quiet:F` causes any skipped lines to be printed. Transposing the output of `matrix()` is necessary because data is read from the file row by row, but stored in the computer column by column. Thus simply `y <- matrix(vecread("myfile3.dat",skip:"#"),5)` would produce a matrix with 5 rows, with each row corresponding to a *column* of the data in the file.

You can also use `vecread()` to read `CHARACTER` data (see also Sec. 7.2). Suppose file `labels.txt` contains

```
Age, Length, Height, Width,   , Strength ! labels for data
```

Then `vecread()` reads a vector of length 6:

```
Cmd> labels <- vecread("labels.txt",character:T); labels
(1) "Age"
(2) "Length"
(3) "Height"
(4) "Width"
(5) ""
(6) "Strength"
```

Note that the two successive commas between `Width` and `Strength` are taken to delimit an empty string `""`. Keywords `skip`, `stop` and `go` work the same as before.

You can use keyword phrase `silent:T` to suppress any warning messages. Thus if the file `starship.txt` looks like

```
Troy     342 67
Tasha    546 53
Beverly  331 49
```

```
Cmd> y <- matrix(vecread("starship.txt",silent:T),2)';y
(1,1)          342          67
(2,1)          546          53
(3,1)          331          49
```

returns the 3 by 2 data matrix without commenting on the row labels.

An alternative method of coping with unreadable items is to use keyword phrase `badvalue:val`, where `val` is a `REAL` scalar or `?`. Any item that does not appear to be a number is read as if it had this value.

```
Cmd> y <- matrix(vecread("starship.txt",badvalue:-99),3)';y
(1,1)          -99          342          67
(2,1)          -99          546          53
(3,1)          -99          331          49
```

The three names could not be read and were replaced by –99. This has the advantage that you can test to see which items were not readable.

See Sec. 7.2 for information on other ways of using `vecread()` to read `CHARACTER` data. See Sec. 7.3 for using keyword `string` to "read" from a `CHARACTER` variable. See Sec. B.6.6, C.5.6 and D.6.6 for using `CONSOLE` as a file name.

2.11.2 **readcols** When a data file such as `myfile3.dat` contains data on several variables, the data for each case on a separate line, you sometimes want to read each variable (column in the file) into a separate MacAnova variable. You often can use pre-defined macro `readcols`, which makes use of `vecread()`, to read such a file and create `REAL` vectors, one for each column in the file. The arguments to `readcols` are the file name, and the *unquoted* names of the variables into which the columns should be placed.

```
Cmd> readcols("myfile3.dat", x1, x2, x3, x4, x5,skip:"#")
```

This creates `REAL` vectors `x1`, `x2`, `x3`, `x4`, and `y` containing the data from columns 1 through 5.

```
Cmd> list(x1,x2,x3,x4,x5)
x1              REAL   6
x2              REAL   6
x3              REAL   6
x4              REAL   6
x5              REAL   6

Cmd> x1 # column (variable) 1
(1)          600      MISSING          522          614          410
(6)      MISSING
```

When the file has a different number of columns from what you expect, `readcols` usually prints an error message.

```
Cmd> readcols("myfile3.dat",x1, x2, x3, x4,skip:"#")# try 4 cols
ERROR: number of rows must divide length of data
```

However, if the number of data elements in the file is actually divisible by the number of variable names you provided, there won't be an error message, but the variables set will be of the wrong length and will not correspond to columns in the file.

```
Cmd> readcols("myfile3.dat", x1, x2, x3, x4, x5, x6,skip:"#")

Cmd> list(x1)
x1              REAL   5    Output from list

Cmd> x1 # x1 has length 5, not 6
(1)          600          644          504          602          644
```

Macro readcols recognizes vecread() keywords stop, skip, quiet, silent and badvalue. See Sec. 2.11.1 for details.

2.11.3 **matread() and read()** A single plain text file can contain several REAL, LOGICAL, CHARACTER or structure data sets, provided each of them has a *header* consisting of at least one line of information specifying a *name* for the data set and its *dimensions* (length for a vector, numbers of rows and columns for a matrix, number of components for a structure). Optional additional header lines can describe the data set and provide formatting information. The file can also contain NULL variables. See Sec. 7.1 for details of the file format. You use matread() or read() to read from such a file. The only difference between them is that matread() prints an warning message if it finds a macro with the specified name instead of a data set and read() does not.

matread() normally requires two CHARACTER variables or quoted strings as arguments, the file name and the data set name. Thus

```
Cmd> x <- matread("data.txt","treedata")
```

searches file data.txt for a data set named treedata, reads it, and assigns the data to variable x. If you omit the data set name, matread() assumes that the first non-empty line in the file is the first header line for the data set. You can create files in the format readable by matread() using matprint() and matwrite() (see Sec. 7.4). If the named file is not in the current default directory or folder, matread() searches the directories or folders in CHARACTER vector DATAPATHS. See Sec. 2.11.6.

File MacAnova.dat distributed with MacAnova contains the data in a form readable by matread(). One of the data sets is halddata, containing data that has often been used as an example when demonstrating regression techniques.

```
Cmd> hald <- matread("macanova.dat","halddata")
halddata     13      5 format
) Hald data from A. Hald, Statistical Theory with Engineering
) Applications, Wiley, New York, 1952, p. 647
) Col. 1: X1 = percent tricalcium aluminate
) Col. 2: X2 = percent tricalcium silicate
) Col. 3: X3 = percent tetracalcium alumino ferrite
) Col. 4: X4 = percent dicalcium silicate
) Col. 5: Y  = cumulative heat evolved from cement hardening after
)              180 days. (calories/gm)
```

The lines following the command line are printed by `matread()` and consist of the name line plus several additional *comment lines* (lines starting with ")") which come before the data. You can suppress the printing of these lines by including `quiet:T` as an additional argument to `matread()`.

Since some commands work only with data vectors rather than matrices, once you have read in a matrix using `matread()`, you may want to split it up into separate vectors, each containing a column of the matrix. Pre-defined macro `makecols` does exactly this. For example, you can create vectors from `data` by

```
Cmd> makecols(hald, x1, x2, x3, x4, y)
```

will create vectors `x1`, `x2`, `x3`, `x4` and `y` from the 5 columns of `hald`.

You can even combine `matread()` and `makecols` in a single expression to create vectors.

```
Cmd> makecols(matread("macanova.dat","halddata",quiet:T),\
x1,x2,x3,x4,y)
```

Ordinarily it is considered an error when `matread()` can't find a data set. However, if you include keyword phrase `notfoundok:T` as an argument, no error will be reported and the value `NULL` is returned. This can be useful in writing macros (see Sec. 9.3) that read data, since it allows you to test whether a data set is available on a file.

```
Cmd> apples <- matread("macanova.dat","apples") # no apples in file
ERROR: dataset or macro apples not found on file macanova.dat

Cmd> list(apples)
WARNING: apples is not defined

Cmd> apples <- matread("macanova.dat","apples",notfoundok:T)

Cmd> list(apples)
apples          NULL

Cmd> isnull(apples)# See Sec. 9.4.2
(1) T
```

Coordinate labels (see Sec. 8.4) can be included along with data sets readable by `matread()` in such a way that they are automatically read (See Sec. 7.1). Whether or not a data set in a file has labels, you can attach labels by using keyword `labels` on `matread()`. Here is an example, reading `halddata`; echoing of the header lines is suppressed by `quiet:T`:

```
Cmd> data <- matread("macanova.dat","halddata",quiet:T,\
labels:structure("@", vector("TricalcAlum","TricalcSi",\
"TetrcalAlFe","DicalcSi","CumulHeat")))

Cmd> data # the label "@" translates into row numbers;see Sec. 8.4.1
      TricalcAlum    TricalcSi  TetrcalAlFe     DicalcSi    CumulHeat
(1)             7           26            6           60         78.5
(2)             1           29           15           52         74.3
(3)            11           56            8           20        104.3
(4)            11           31            8           47         87.6
(5)             7           52            6           33         95.9
(6)            11           55            9           22        109.2
```

```
(7)      3      71      17       6      102.7
(8)      1      31      22      44       72.5
(9)      2      54      18      22       93.1
(10)    21      47       4      26      115.9
(11)     1      40      23      34       83.8
(12)    11      66       9      12      113.3
(13)    10      68       8      12      109.4
```

2.11.4 **getdata** Macro `getdata` is designed make it easier to work with "libraries" of data sets in the form readable by `matread()`. It has a single *unquoted* argument that is interpreted as the name of a data set on the file whose name is the value of `CHARACTER` variable `DATAFILE`. When MacAnova is started up, `DATAFILE` is pre-defined to be `"macanova.dat"`, but you may assign a new value at any time. `getdata` uses `matread()` to read the file. For example, using the default value of `DATAFILE`,

```
Cmd> hald <- getdata(halddata)
```

would be equivalent to `hald <- matread("macanova.dat","halddata")`. To use `getdata` to retrieve `treedata` from file `data.txt`, you would need the following:

```
Cmd> DATAFILE <- "data.txt"; x <- getdata(treedata)
```

Subsequent use of `getdata` would continue to retrieve data from `data.txt`.

If the file named in `DATAFILE` is not in the default directory or folder or in one of the directories or folders specified by `CHARACTER` vector `DATAPATHS` (see Sec. 2.11.6), it should be a complete "path name", such as `"C:/MACANOVA/MACANOVA.DAT"` or `"Macintosh Hard Disk:MacAnova:MacAnova.dat"`. See Appendices B through F. Note that on DOS/Windows computers you may use "`/`" instead of "\" in path names. In fact, if you want to use "\" you have to use "\\" (see Sec. 2.5).

2.11.5 **inforead()** As exemplified in Sec. 2.11.3, data sets on external files that are readable by `matread()` may have associated comment lines with information about the data (see also Sec. 7.1). The same is true of macros on external files (Sec. 7.5). It can be helpful to save these comments in a `CHARACTER` variable so that they are instantly available for reference. They can also be attached to the variable itself as an informative *note* (see Sec. 8.9). You can do this using `inforead()` which is used much the same as `matread()` and `read()`.

`inforead(FileName,Name)` searches file `FileName` for a macro or data set with name `Name`. If found, `inforead()` reads the comments (the lines starting with "`)`") following the header line and returns a `CHARACTER` variable containing these lines, with the leading "`)`" stripped off. `FileName` and `Name` must be quoted strings or `CHARACTER` variables. The actual contents of the data set or macro are ignored and there is no checking as to whether the header line is in correct format.

`inforead(FileName)` does the same for the first data set or macro on the file, assuming that line 1 is the header line.

In versions with windows (Macintosh, Windows, Motif), if `FileName` is `""`, you will be able to select the file using a dialog box.

`inforead(FileName[,Name],quiet:T)` returns the comments, but suppresses any

echoing of the header and comment lines of the data set or macro.

```
Cmd> info <- inforead("macanova.dat","halddata", quiet:T)

Cmd> info # see Sec. 2.11.3.
(1) " Hald data from A. Hald, Statistical Theory with Engineering
 Applications, Wiley, New York, 1952, p. 647
 Col. 1: X1 = percent tricalcium aluminate
 Col. 2: X2 = percent tricalcium silicate
 Col. 3: X3 = percent tetracalcium alumino ferrite
 Col. 4: X4 = percent dicalcium silicate
 Col. 5: Y  = cumulative heat evolved from cement hardening after
              180 days. (calories/gm)"

Cmd> attachnotes(data,info)# attach info to data itself (Sec. 8.9.1)

Cmd> getnotes(data) # see Sec. 8.9.1
(1) " Hald data from A. Hald, Statistical Theory with Engineering
 Applications, Wiley, New York, 1952, p. 647
 Col. 1: X1 = percent tricalcium aluminate
 Col. 2: X2 = percent tricalcium silicate
 Col. 3: X3 = percent tetracalcium alumino ferrite
 Col. 4: X4 = percent dicalcium silicate
 Col. 5: Y  = cumulative heat evolved from cement hardening after
              180 days. (calories/gm)"
```

inforead(FileName,Name[,quiet:T],notfoundok:T) behaves identically when the data set or macro is found. However, when it is not found, no message is printed and NULL is returned. When used in a macro, this feature allows special action if Name is not found.

```
Cmd> applesinfo <- matread("macanova.dat","apples",notfoundok:T)

Cmd> list(applesinfo)
applesinfo      NULL
```

2.11.6 **HOME, DATAPATHS and adddatapath** When MacAnova starts up, two CHARACTER variables, HOME and DATAPATHS, are automatically created to make it easier for MacAnova to find files. On all systems except Unix, these are both initialized to the complete "path name" of the directory or folder where MacAnova is located. On a Unix system, HOME is initialized to the name of the user's home directory (environmental variable $HOME) and DATAPATHS is initialized to the name of an installation-dependent directory containing data files. Both HOME and DATAPATHS can be modified once MacAnova has started up, perhaps in a start up file (See Sec. 7.8). In particular, you can add other directory or folder names to DATAPATHS, making it a vector.

If a file name is specified as a simple file name, such as "halddata", with no special "path" characters such as ":" or "/", Macanova first attempts to read it in whatever the current default directory or folder is. If that is not successful, it makes an attempt in directory DATAPATHS[1]; if not successful there, it looks in DATAPATHS[2], and so on, giving up only if it is not found in any directory or folder in DATAPATHS. Here is an example of its use. File Hald is not in the default directory/folder or in DATAPATHS[1] but is in a sub-directory or folder.

```
Cmd> DATAPATHS # one folder name in DATAPATHS
(1) "Macintosh HD:MacAnova Folder:"

Cmd> y <- vecread("Hald") # can't find file Hald there
ERROR: vecread cannot open file Hald

Cmd> DATAPATHS <- vector("Macintosh HD:MacAnova Folder:Data",\
DATAPATHS) # add new folder name at start of DATAPATHS

Cmd> y <- vecread("Hald"); y[run(5)] # Found
(1)           7           26            6           60         78.5
```

Vector DATAPATHS can be of any length, providing lots of places to look for a file. Pre-defined macro adddatapath makes it easier to add a folder or directory name to either the beginning or end of DATAPATHS

```
Cmd> adddatapath("timeser"); DATAPATHS # add "timeser" at start
(1) "timeser"
(2) "Macintosh HD:MacAnova Folder:Data:"
(3) "Macintosh HD:MacAnova Folder:"

Cmd> adddatapath("mvdata",T);DATAPATHS # add "mvdata" at end
(1) "timeser"
(2) "Macintosh HD:MacAnova Folder:Data:"
(3) "Macintosh HD:MacAnova Folder:"
(4) "mvdata"
```

These examples use the path separating character ":" appropriate for a Macintosh. On other computers you would use names like "D:/MACANOVA/DATA/" (DOS/Windows) or "/users/kb/macanova/data/" (Unix).

Variable HOME comes into play when you use a file name of the form "~:name" (on a Macintosh) or "~/name" (on other computers). MacAnova looks for the file in the directory whose name is in HOME.

```
Cmd> HOME
(1) "Macintosh HD:MacAnova Folder:"

Cmd> y <- vecread("~:Data:Hald") # found
```

You can change HOME to be the name of any folder or directory.

```
Cmd> HOME <- "Macintosh HD:MacAnova Folder:Data:"

Cmd> y <- vecread("~:Hald") # found
```

2.12 **Simple statistics** While MacAnova is oriented towards the analysis of variance, multivariate analysis and time series analysis, it provides many simpler statistical functions as well. In addition to the functions and macros described here, function tab() allows cross tabulation of data by the levels of integer valued factors. See Sec. 2.12.

2.12.1 **describe()** When you want simple descriptive statistics of data in variable x, use describe(x). The default use of describe() computes, for each column of x, the number of non-missing values, the mean, variance, median, maximum, minimum, and upper and lower quartiles. Using keyword phrases, additional statistics can be computed including the standard deviation, g_1 and g_2 (indices of skewness and

kurtosis) and moments m_2, m_3 and m_4 about the mean.

The output of `describe()` is a structure with a component for each statistic computed. For the default use of `describe()`, the components are `n`, `min`, `q1`, `median`, `q3`, `max`, `mean`, and `var`.

```
Cmd> x <- matrix(vector(2,?,4,5,8, 2,3,4,1,7, 8,4,3,6,5),5); x
(1,1)            2            2            8
(2,1)      MISSING            3            4
(3,1)            4            4            3
(4,1)            5            1            6
(5,1)            8            7            5

Cmd> describe(x)
WARNING: missing values in input to describe
WARNING: output reflects nonmissing data only
component: n       Number of non-missing values
(1)              4            5            5
component: min     Minimum value
(1)              2            1            3
component: q1      Lower quartile
(1)              3            2            4
component: median  2nd quartile
(1)            4.5            3            5
component: q3      Upper quartile
(1)            6.5            4            6
component: max     Maximum value
(1)              8            7            8
component: mean    Average
(1)           4.75          3.4          5.2
component: var     Variance
(1)           6.25          5.3          3.7
```

If you don't want all the statistics, you can specify the ones you want using keywords matching the component names.

```
Cmd> describe(x,mean:T,var:T)
WARNING: missing values in input to describe
WARNING: output reflects non-missing data only
component: mean
(1)           4.75          3.4          5.2
component: var
(1)           6.25          5.3          3.7
```

If you specify only one statistic (`describe(x,mean:T)`) you get a scalar or a vector, not a structure.

If `x` is a structure, `describe(x)` computes the statistics for each component. In that case, each component of `describe(x)` is itself a structure similar to `x`:

```
Cmd> describe(temperatures, mean:T,var:T)
component: mean
  component: Saturday
(1)         77.6
  component: Sunday
(1)         76.6
  component: Monday
(1)       71.286
component: var
  component: Saturday
(1)       138.49
  component: Sunday
(1)         96.8
  component: Monday
(1)       150.68
```

Here `temperatures` is the structure used as an example in Sec. 2.8.16.

There are additional statistics that may be computed only using keyword phrases.

Keyword / component name	Statistic
stddev	$\sqrt{k_2} = \sqrt{\sum (x-\bar{x})^2/(n-1)}$
m2	$\sum (x-\bar{x})^2/n$
m3	$\sum (x-\bar{x})^3/n$
m4	$\sum (x-\bar{x})^4/n$
g1	$k_3/k_2^{3/2}$
g2	k_4/k_2^2

k_2, k_3 and k_4 are Fisher's k-statistics defined as

$$k_2 = \frac{S_2}{n-1}, k_3 = \frac{nS_3}{(n-1)(n-2)}, k_4 = \frac{n(n+1)S_4 - 3(n-1)S_2^2}{(n-1)(n-2)(n-3)}$$

where $S_\ell = \sum (x-\bar{x})^\ell$.

g_1 and g_2 are not identical to $\sqrt{b_1} = m_3/m_2^{3/2}$ and $b_2 = m_4/m_2^2 - 3$ which are also often used to measure skewness and kurtosis.

Expressed in terms of m_2, m_3 and m_4 or $\sqrt{b_1}$ and b_2

$$g_1 = \sqrt{\frac{n(n-1)}{n-2}}\frac{m_3}{m_2^{3/2}} = \sqrt{\frac{n(n-1)}{n-2}}\sqrt{b_1}$$

$$g_2 = \frac{n^2-1}{(n-2)(n-3)}\left(\frac{m_4}{m_2^2} - 3 + \frac{6}{n+1}\right) = \frac{n^2-1}{(n-2)(n-3)}\left(b_2 + \frac{6}{n+1}\right)$$

When $n \le 2$, g_1 is computed to be 0. When $n \le 3$, g_2 is computed to be 0.

```
Cmd> describe(x, m2:T, m3:T, m4:T, g1:T, g2:T)
WARNING: missing values in input to describe
WARNING: output reflects non-missing data only
component: m2
(1)       4.6875         4.24         2.96
component: m3
(1)       3.2812        6.048        2.016
component: m4
(1)        42.27       41.027       17.475
component: g1
(1)         0.56       1.0327      0.59013
component: g2
(1)        0.928       1.1285    -0.021914
```

g_1 and g_2 are sometimes used to test the null hypothesis that a sample comes from a normal population. If the data are a random sample from a normal distribution, then g_1 and g_2 have mean 0 and variances

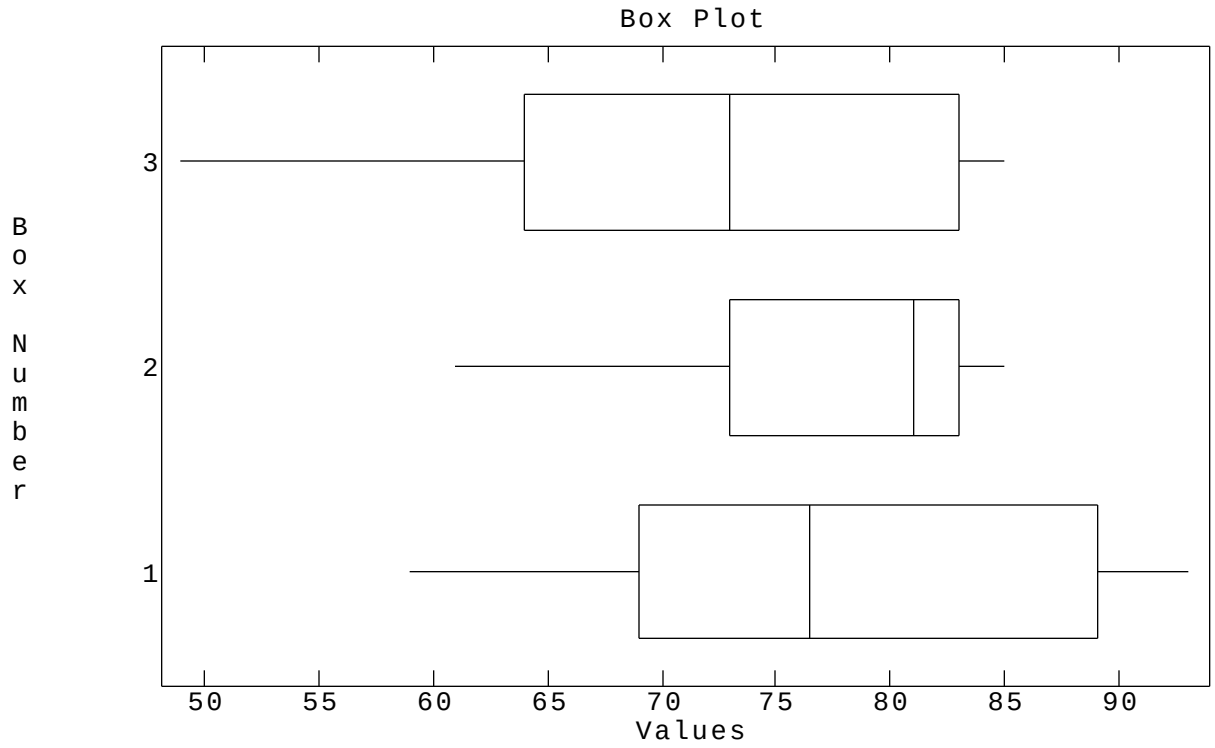
$$V[g_1] = \frac{6n(n-1)}{(n-2)(n+1)(n+3)}, \quad V[g_2] = \frac{24n(n-1)^2}{(n-3)(n-2)(n+2)(n+5)}$$

2.12.2 **boxplot(), vboxplot, stemleaf() and hist** These are intended to give a quick look at the distribution of one or more samples of data.

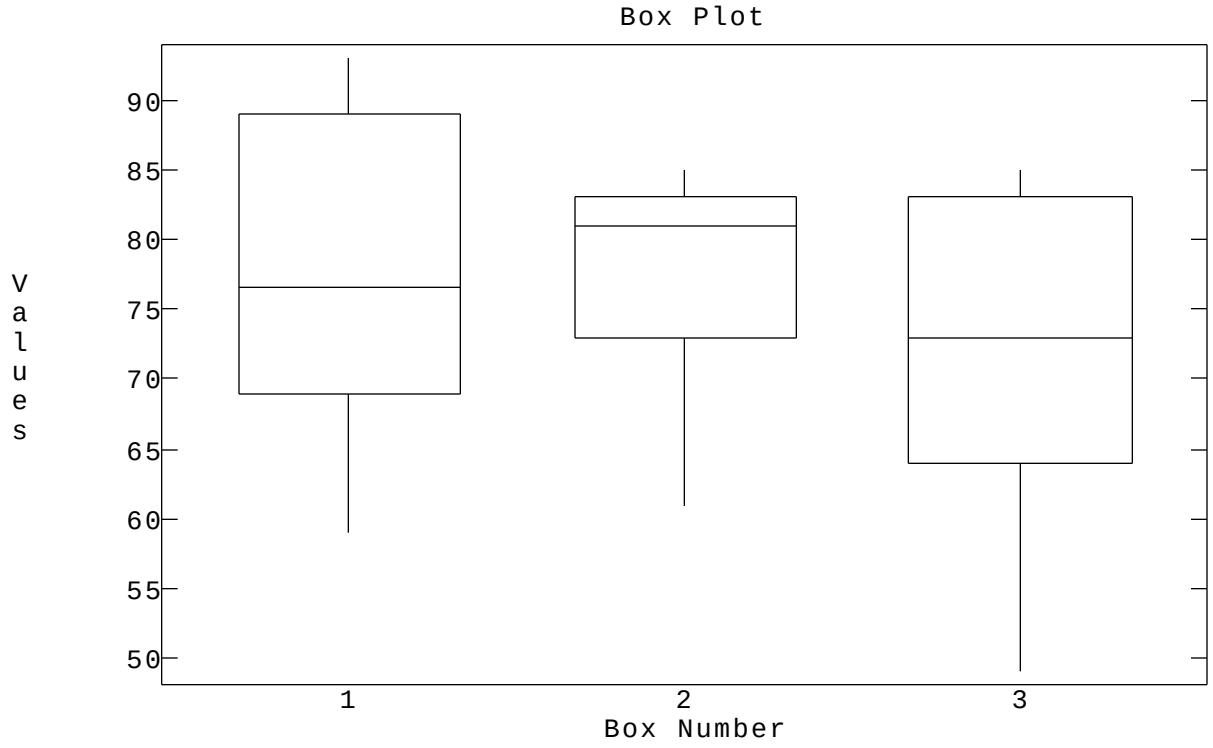
`boxplot(x)` and pre-defined macro `vboxplot` draw a Tukey box and whisker diagram, often called a *box plot*, of the data in `x`. This is a graphical summary of the distribution of the values. When `x1`, `x2`, ..., `xk` are several `REAL` vectors, `boxplot(x1,x2,...,xk)` draws parallel horizontal box plots, one box per vector. This can be a very useful way to compare the distributions of the variables. If you prefer a plot box plot in which the boxes are oriented *vertically*, use `boxplot(x,vertical:T)` or more simply, `vboxplot(x1,x2,...,xk)`.

A box plot consists of a central box, with a line drawn somewhere between the left (lower) and right (upper) ends, "whiskers" extending off the ends of the box, and separately plotted moderate and extreme "outliers". The ends of the box are drawn at the lower and upper quartiles of the data, respectively, and the middle line is the median. Whiskers extend to the furthest (lowest and highest) data values that are still within the "inner fences", lower quartile – 1.5 IQR and upper quartile + 1.5 IQR (IQR = inter-quartile range). Points outside the "outer fences", lower quartile – 3 IQR and upper quartile + 3 IQR, are plotted as "`o`" and points between the inner and outer fences are plotted as "`*`". Points outside the inner fences may be tentatively identified as outliers.

```
Cmd> boxplot(temperatures$Saturday, temperatures$Sunday,\
temperatures$Monday)
```




```
Cmd> vboxplot(temperatures$Saturday, temperatures$Sunday,\
temperatures$Monday) # same with vertical orientation
```



The bottom or left box represents the Saturday temperatures, the middle box, the Sunday temperatures, and so on. No temperatures were outliers so there are no

symbols beyond the whiskers.

```
Cmd> boxplot(temperatures$Saturday, temperatures$Sunday,\
temperatures$Monday,vertical:T)
```

is an alternative way to get vertically oriented boxes.

When `str` is a structure whose components are REAL vectors, `boxplot(str)` draws parallel box plots for each component. Thus, since `temperatures` is a structure, `boxplot(temperatures)` would produce the same display of box plots.

A common way to create a structure for use with `boxplot()` is to use `split()` (see Sec. 9.1.1). In the following, `temps` is a vector combining the data from the three components of `temperatures`, and a `day` is a vector of integers coding Saturday as 1, Sunday as 2 and Monday as 3

```
Cmd> print(temps,day,format:"2.0f") #see Sec. 7.4 for use of format
temps:
 (1) 65 71 75 86 91 93 89 78 69 59 61 73 85 83 81 51 65 71 78 83 84
(22) 85 84 81 75 69 64 59 49
day:
 (1)  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  3  3  3  3  3  3
(22)  3  3  3  3  3  3  3  3

Cmd> split(temps,day) # create structure, splitting up temps by day
component: day1
(1)           65          71          75          86          91
(6)           93          89          78          69          59
component: day2
(1)           61          73          85          83          81
component: day3
 (1)           51          65          71          78          83
 (6)           84          85          84          81          75
(11)           69          64          59          49

Cmd> boxplot(split(temps,day))
```

This would produce the same horizontally oriented plot as before.

You can print a stem and leaf display of the data in a REAL vector or a n by 1 matrix `x` by `stemleaf(x)`:

```
Cmd> stemleaf(temperatures$Monday)
    1    4.|9
    2    5*|1
    3    5.|9
    4    6*|4
    6    6.|59
    7    7*|1
    7    7.|58
    5    8*|1344
    1    8.|5

         1*|1  represents 11  Leaf digit unit = 1
```

The numbers to the left of "|" are the "stems" and the numbers to the right of "|" are "leaves". The stems are taken from the first digit (sometimes the first 2 digits) of each

number. There is one single digit leaf for every number in the data set. The value of each leaf is the digit following the stem in the corresponding number. Because the values of Monday temperatures range only from 49 to 85, here MacAnova uses "half digit" stems whose leaves are either 0, 1, 2, 3 or 4 (for stems labelled 5*, 6*, ...) or 5, 6, 7, 8 or 9 (for stems labelled 4., 5., ...).

You can control the number of stems stemleaf() will use by an optional second argument which allows you to set the maximum number of stems.

```
Cmd> stemleaf(temperatures$Monday,5) # use at most 5 stems
    1     4|9
    3     5|19
    6     6|459
  ( 3)    7|158
    5     8|13445

          1|1  represents 11  Leaf digit unit = 1
```

The first column printed contains the "depth" – cumulative counts from each "tail" of the distribution, plus, in parentheses, the number of leaves on the stem which includes the median. Thus, there are 5 cases with values 80 and above, 6 with values less than 70 and 3 values in the 70's. The leaves are always computed by rounding toward 0, so that a temperature of 79.9 would show up as a leaf of 9 on stem 7 and would not be rounded up to 80 (leaf of 0 on stem of 8).

If you don't want the depth column, use keyword phrase depth:F.

It's sometimes helpful to have a few summary statistics printed with a stem and leaf display. Keyword phrase stat:T causes extremes and quartiles to be printed. Here is an example with no depth column and with the extremes and quartiles:

```
Cmd> stemleaf(temperatures$Saturday,5,depth:F,stat:T)
n=10, Min=59, Q1=69, M=76.5, Q3=89, Max=93
    5|9
    6|59
    7|158
    8|69
    9|13

    1|1 represents 11  Leaf digit unit = 1
```

Potential outliers in a stem and leaf diagram are defined the same way as for a box plot, namely as values outside the "inner fences." stemleaf() does not ordinarily include outliers as leaves, but lists them separately. Let's make the first Monday temperature an outlier.

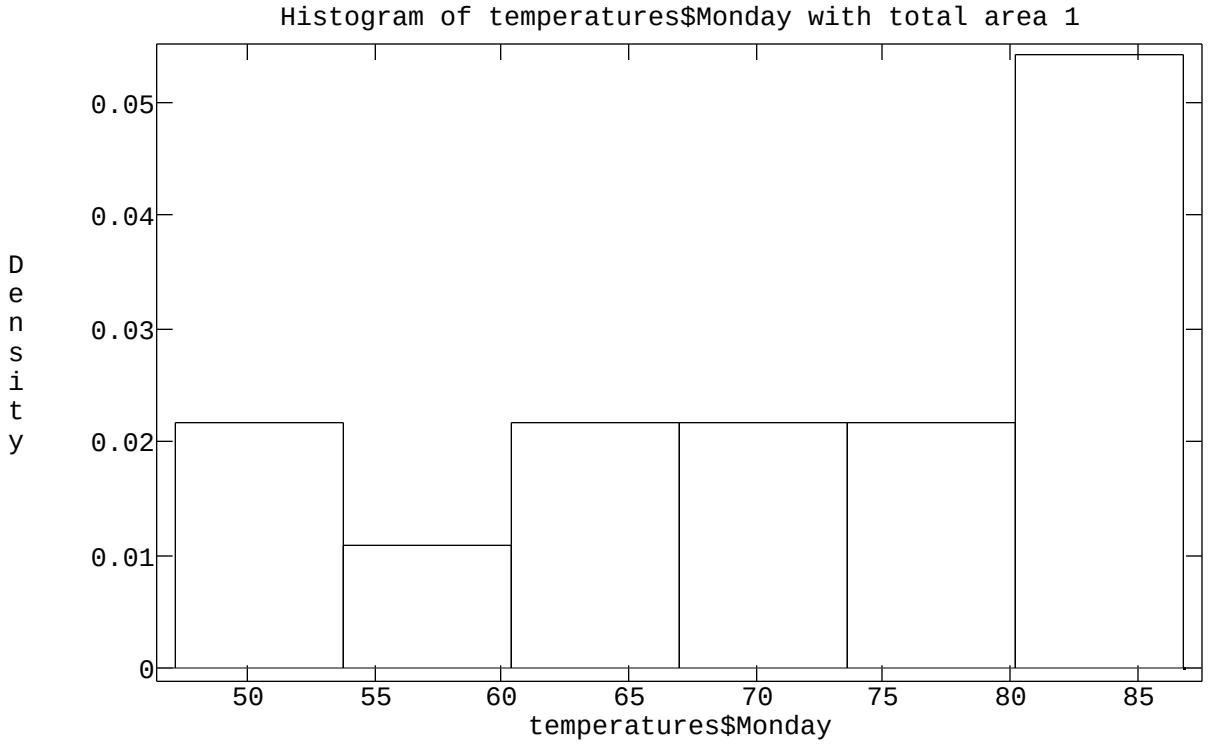
```
Cmd> monday <- temperatures$Monday

Cmd> monday[1] <- 120; stemleaf(monday,5)
    1     4|9
    2     5|9
    5     6|459
  ( 3)    7|158
    6     8|13445
 High  120
          1|1  represents 11  Leaf digit unit = 1
```

If you want to outliers to be represented as leaves, use stemleaf(x,outliers:F).

A histogram is a more conventional way to display the distribution of data points. You can draw histograms using `hist`, a pre-defined macro. If you are satisfied to let MacAnova pick the number of bars in the histogram, just type `hist(x)`, where `x` is a `REAL` vector. If that's not good enough, `stemleaf(x,n)` draws a histogram with `n` bars, where `n` is a positive integer. The class limits (bar edges) are chosen so that the bars all have the same width arbitrarily but will not normally by "neat". The bar heights are in the so called *density scale* – that is for each bar, width × height = the proportional of values falling in the bar.

```
Cmd> hist(temperatures$Monday,6) # draw histogram with 6 bars
```



You can, in fact, completely specify the class limits. For example,

```
Cmd> hist(temperatures$Monday,vector(40,50,60,70,80,85,90))
```

draws a histogram with unequally spaced class limits 40, 50, 60, 70, 80, 85 and 90.
You can specify a title and X-axis labels for a histogram or for a box plot using keywords `title` and `xlab`. Examples might be

```
Cmd> boxplot(temperatures,\
title:"Saturday through Monday Temperatures",\
xlab:"Degrees Fahrenheit")

Cmd> hist(temperatures$Monday,vector(40,50,60,70,80,85,90),\
title:"Monday temperatures",xlab:"Degrees Fahrenheit")
```

Normally, `hist` assigns a value exactly equal to a class limit to the bar *to its left*, that is the right boundary point is considered to be in a bar but not the left.. If you include `leftendin:T` as an argument to `hist`, the left boundary is considered to be in the bar, but not the right, so that such a value is assigned to the bar to its right. This

corresponds to a convention used in some statistics text books.

2.12.3 **sort(), rank(), grade(), rankits(), and halfnorm()** These functions all have to do with the ordering of data values. They expect a single vector or matrix argument x plus an optional keyword phrase down:T. All work with REAL data; sort(), rank() and grade() also work with CHARACTER data. When x is a matrix, they all operate on each column separately.

For CHARACTER data, ordering is in alphabetical order using the ASCII collating sequence in which most punctuation and all numerals sort ahead of upper case letters which sort ahead of lower case letters. A space sorts ahead of all printable characters. Here is the explicit ordering starting with space:

Sorting order of characters
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~

A null string "" sorts before any other string.

sort(x) returns a vector or matrix, each column of which is the corresponding column of x arranged in increasing or alphabetical order. sort(x,down:T) orders each column in decreasing or reverse alphabetical order. In both cases, when x is REAL, MISSING values are moved to the end.

rank(x) computes the ranks of the non-missing data in each column of x, with the smallest or alphabetically first value assigned rank 1. rank(x,down:T) does the same, except the largest or alphabetically last value assigned rank 1. For REAL data, the rank of a MISSING value is set MISSING.

By default, when there are ties in REAL data, the rank computed for all the elements in a set of equal (tied) elements is the average of their ranks. Thus rank(vector(1.2, 3.4,1.2,5,3.4,3.4)) is vector(1.5,4,1.5,6,4,4). For CHARACTER data, the ranks associated with tied elements are distinct and unpredictable. See below for an alternative treatment of ties with REAL data.

grade(x) computes what might be called inverse ranks of the columns of x. When x is a vector, after j <- grade(x), j[1] is the index (row number) of the smallest or alphabetically first element of x, j[2] is the index of the second smallest, and so on. If length(x) is n, then grade(x)[n] is the index of the largest number (maximum or last in alphabetical order). Thus x[grade(x)] is the same as sort(x). x[grade(x[,1]),] returns a matrix the same shape as x containing the rows of x rearranged so that the first column is in increasing order. This is probably the most important application of grade(). j <- grade(x,down:T) does the same, except that j[1] is the index of the largest value or last in alphabetical order. When x is a matrix, grade(x) processes each column separately. If x is REAL, the indices of any MISSING values are always put in the last rows of grade(x) and grade(x,down:T).

Function rankits() computes the *normal scores* or *rankits* of the non-missing elements in each column and sets the rankit of a MISSING value to MISSING. When x

is a vector and there are no ties, `rankits(x)[i]` is the normal probability point z_{p_i}, where p_i = `(rank(x)[i] - .375)/(n + .25)`. When there are ties, `rankits()` breaks them arbitrarily. Note that the order of the elements of `x` is retained.

Function `halfnorm()` is similar to `rankits()` except it computes *half normal scores*, that is, when there are no ties, `halfnorm(x)[i]` is z_{p_i} where

p_i = `.5 + .5*(rank(abs(x))[i] - .375)/(n + .25))`.

```
Cmd> x <- matrix(vector(2,?,4,5,8, 2,3,4,1,7, 8,4,3,6,5),5); x
(1,1)            2            2            8
(2,1)      MISSING            3            4
(3,1)            4            4            3
(4,1)            5            1            6
(5,1)            8            7            5

Cmd> sort(x)
WARNING:  MISSING values in argument to sort
(1,1)            2            1            3
(2,1)            4            2            4
(3,1)            5            3            5
(4,1)            8            4            6
(5,1)      MISSING            7            8  MISSING at end

Cmd> rank(x)
WARNING:  MISSING values in argument to rank
(1,1)            1            2            5
(2,1)      MISSING            3            2
(3,1)            2            4            1
(4,1)            3            1            4
(5,1)            4            5            3

Cmd> rankits(x)
WARNING:  MISSING values in argument to rankits
(1,1)      -1.0491      -0.4972       1.1798
(2,1)      MISSING            0      -0.4972
(3,1)     -0.29931       0.4972      -1.1798
(4,1)      0.29931      -1.1798       0.4972
(5,1)       1.0491       1.1798            0

Cmd> grade(x)
WARNING:  MISSING values in argument to grade
(1,1)            1            4            3
(2,1)            3            1            2
(3,1)            4            2            5
(4,1)            5            3            4
(5,1)            2            5            1

Cmd> sort(x,down:T)
WARNING:  MISSING values in argument to sort
(1,1)            8            7            8
(2,1)            5            4            6
(3,1)            4            3            5
(4,1)            2            2            4
(5,1)      MISSING            1            3  MISSING at end
```

```
Cmd> rank(x,down:T)
WARNING:  MISSING values in argument to rank
(1,1)            4            4            1
(2,1)      MISSING            3            4
(3,1)            3            2            5
(4,1)            2            5            2
(5,1)            1            1            3

Cmd> grade(x,down:T)
WARNING:  MISSING values in argument to grade
(1,1)            5            5            1
(2,1)            4            3            4
(3,1)            3            2            5
(4,1)            1            1            2
(5,1)            2            4            3
```

With any of these functions, if x is an array, the result is an array of the same size and shape, with the function being applied to every vector specified by fixed values of subscripts 2, 3 ... .

```
Cmd> ary <- array(vector(27.3,20.8,32.0, 28.8,18.9,28.1,\
22.9,32.4,32.1,  31.4,29.1,31.3), 3, 2, 2)# 3 by 2 by 2

Cmd> ary[,2,2]
(1,1,1)         31.4
(2,1,1)         29.1
(3,1,1)         31.3

Cmd> sort(ary)[,2,2] # same as sort(ary[,2,2])
(1,1,1)         29.1
(2,1,1)         31.3
(3,1,1)         31.4
```

You can also use any of these functions when x is a structure (Sec. 2.8.16), whose non-structure components all have the same type, either REAL or CHARACTER. In that case, each function returns a structure of the same form, each of whose non-structure components is the result of applying the function to the corresponding component of x.

```
Cmd> dryandwet <- structure(dry:vector(3.32,2.99,1.61,2.52),\
wet:vector(3.60,4.21,3.63)); dryandwet
component: dry
(1)          3.32         2.99         1.61         2.52
component: wet
(1)           3.6         4.21         3.63

Cmd> sort(dryandwet)
component: dry
(1)          1.61         2.52         2.99         3.32
component: wet
(1)           3.6         3.63         4.21
```

You can use keyword phrase ties:method to change the the way rank(), rankits(), and halfnorm() handle tied REAL data. method must be one of "average", "minimum" or "ignore" which you can abbreviate as "a", "b" and "i". Suppose *k* elements in a vector (column) are tied, that is they all have the same value and no other element has this value, and suppose the ranks these elements would have if

their values were very slightly changed so as to break the ties while preserving other ordering would be r, r+1, r+2, ..., r+k–1. The following table describes the ranks computed by `rank()` for the tied values for each of the three possible methods.

Value for ties	Computed ranks
`"average"`	All ranks = $(r+(r+1)+...+(r+k-1))/k = r + (k-1)/2$
`"minimum"`	All ranks = r
`"ignore"`	r, r+1, r+2, ..., r+k–1 in an unpredictable order

`rankits(x,ties:method)` and `halfnorm(x,ties:method)` are equivalent to finding the normal probability points corresponding to p_i = `(rank(x,ties:method)[i] - .375)/(n + .25)` and p_i = `.5 + .5*(rank(abs(x),ties:method)[i] - .375)/(n + .25))`, respectively. It is hard to imagine a situation in which `ties:"minimum"` would be useful with `rankits()` and `halfnorm()`.

```
Cmd> rank(vector(3,2,2,1,3),ties:"average") # default
(1)          4.5          2.5          2.5            1          4.5

Cmd> rank(vector(3,2,2,1,3),ties:"minimum")
(1)            4            2            2            1            4

Cmd> rank(vector(3,2,2,1,3),ties:"ignore")
(1)            4            2            3            1            5

Cmd> rankits(vector(3,2,2,1,3),ties:"average")
(1)      0.79164     -0.24104     -0.24104      -1.1798      0.79164

Cmd> rankits(vector(3,2,2,1,3),ties:"ignore") # default
(1)       0.4972      -0.4972            0      -1.1798       1.1798
```

2.12.4 **sum(), prod(), max(), min()** These functions compute summary values for a REAL or LOGICAL vector or for each column of a REAL or LOGICAL matrix. For LOGICAL data, True is interpreted as 1 and False as 0.

```
Cmd> sum(x) # same x as was used in Sec. 2.12.3
WARNING: MISSING values found by sum
(1,1)           19           17           26 Sums down columns

Cmd> prod(x)
WARNING: MISSING values found by prod
(1,1)          320          168         2880 Product down cols

Cmd> min(x)
WARNING: MISSING values found by min
(1,1)            2            1            3 Minimum value in col

Cmd> max(x)
WARNING: MISSING values found by max
(1,1)            8            7            8 Maximum value in col

Cmd> sum(x <= 4) # count the numbers <= 4 in each column
WARNING: comparison with missing value(s) near sum(x <= 4)
WARNING: MISSING values found by sum
(1,1)            2            4            2
```

When there are no MISSING values, `sum(x)/nrows(x)` computes the column means

of a REAL matrix x, producing a row vector, that is a matrix with 1 row.

With any of these functions, when the argument is an array, the result is an array with the same number of dimensions but with the first dimension having length 1. The function is applied to every vector specified by fixed values of subscripts 2, 3 ... . We illustrate this use with the array ary created in Sec. 2.12.3.

```
Cmd> sum(ary) # sum over first dimension.
(1,1,1)         80.1         87.4
(1,2,1)         75.8         91.8

Cmd> max(ary) # maximimum over first dimensions
(1,1,1)           32         32.4
(1,2,1)         28.8         31.4
```

These functions also accept a list of REAL or LOGICAL scalars or vectors as arguments. For example, sum(1,vector(3,5)) is equivalent to sum(vector(1,3,5)).

If an argument to these functions is NULL, it is ignored. If all the arguments are NULL, the result is NULL.

```
Cmd> sum(NULL,1)
(1)             1

Cmd> @a <- max(NULL, NULL); list(@a)
@a              NULL
```

Finally, any of these functions can take a structure, all whose components are either REAL or LOGICAL.

```
Cmd> sum(dryandwet)
component: dry
(1)         10.44
component: wet
(1)         11.44
```

2.12.5 **Computing correlations – cor()** If x is a data matrix, cor(x) computes the its correlation matrix, treating each column as a variable. If x is *n* by *m* , cor(x) is *m* by *m* , with 1's down the diagonal and cor(x)[i,j] the Pearson correlation between x[,i] and x[,j].

```
Cmd> w <- matrix(vector(45.5,42.1,53.8,48.5,44.5,\
58.4,74.1,72.0,63.8,67.7,  28.7,35.9,32.1,28.5,28.1),5)

Cmd> w # 3 columns
(1,1)         45.5         58.4         28.7
(2,1)         42.1         74.1         35.9
(3,1)         53.8           72         32.1
(4,1)         48.5         63.8         28.5
(5,1)         44.5         67.7         28.1

Cmd> cor(w) # 3 by 3 matrix
(1,1)            1     0.049932     -0.16287
(2,1)     0.049932            1      0.78611
(3,1)     -0.16287      0.78611            1
```

If x1, x2, ... are vectors or matrices all with the same number of rows, then

`cor(x1,x2,x3,...)` is equivalent to `cor(hconcat(x1,x2,x3,...))` (see Sec. 2.10.6).

```
Cmd> makecols(w,w1,w2,w3) # See Sec. 2.11.3;

Cmd> cor(w1,w2,w3) # same as cor(w)
(1,1)            1     0.049932     -0.16287
(2,1)     0.049932            1      0.78611
(3,1)     -0.16287      0.78611            1
```

You can use subscripts (Sec. 2.8.14) to get the correlation between two variables as a single number:

```
Cmd> cor(w1,w2)[1,2] # compute a single correlation coef
(1,1)     0.049932
```

If there are any `MISSING` values in the arguments, correlations are computed using only complete cases, effectively deleting any row with any `MISSING` values.

```
Cmd> ww <- w; ww[2,3] <- ? # put a MISSING value in row 2

Cmd> cor(ww) # same as cor(w[-2,])
WARNING: 1 cases with missing values deleted in cor
(1,1)            1      0.64695      0.92817
(2,1)      0.64695            1      0.65979
(3,1)      0.92817      0.65979            1
```

2.12.6 **Student's *t* related functions and macros – tval(), t2val(), tint(), t2int() and twotailt** There are four functions for statistical analyses based on Student's *t*. Functions `tval()` and `t2val()` calculate one- and two-sample *t*-statistics; `tint()` and `t2int()` compute confidence intervals for the mean of a single sample and the difference between the means of two independent samples, respectively. Macro `twotailt` computes a two-tail *P* value for a *t*-statistic.

`tval(x)`, where `x` is a `REAL` vector, computes the usual one-sample *t*-statistic $\frac{\bar{x}}{s_x / \sqrt{n}}$ to test the null hypothesis H_0: $\mu_x = 0$. You can test a different null hypothesis, say H_0: $\mu_x = 100$ by the statistic computed by `tval(x-100)`.

When `x` and `y` are `REAL` vectors of the same length, `tval(x-y)` computes the *paired* *t*-statistic which you can use to test the null hypothesis H_0:$\mu_x = \mu_y$.

```
Cmd> x1 <- vector(3,2,5,4,6,8,6,4,3,7,3)

Cmd> vector(tval(x1), tval(x1 - 5)) # test H0: μ = 0 and H0 μ = 5
(1)       8.0437     -0.63088  1-sample t-statistics

Cmd> x2 <- vector(4,2,5,6,7,9,7,4,3,8,5); tval(x2-x1)
(1)       3.6145                Paired t-statistic
```

When `x` and `y` are `REAL` vectors, `t2val(x,y)` computes the *two-sample* *t*-statistic $\frac{\bar{x} - \bar{y}}{\sqrt{\frac{1}{n_1} + \frac{1}{n_2}} s_p}$ where $s_p^2 = \frac{(n_1 - 1)s_x^2 + (n_2 - 1)s_y^2}{n_1 + n_2 - 2}$ is a pooled estimate of variance based on the two sample variances s_x^2 and s_y^2 with $n_1 - n_2 - 2$ degrees of freedom. This is often used to test the null hypothesis H_0: $\mu_x = \mu_y$ or, equivalently H_0: $\mu_x - \mu_y = 0$. To test a different null hypothesis, say H_0: $\mu_x - \mu_y = 10$, use `t2val(x-10,y)`. This use of

`t2val()` assumes the samples come from populations with equal variances ($\sigma_x^2 = \sigma_y^2$). See below for the use of `t2val()` when you do not assume equal variances.

```
Cmd> y <- vector(7,9,6,5,7,6,9,8)
Cmd> # test H0:μx = μy and H0:μx-μy = -3
Cmd> vector(t2val(x1,y), t2val(x1-(-3),y))
(1)      -3.0795       0.63278  2-sample t-statistics
```

You can compute two-tail *P* values for Student's *t* statistics using pre-defined macro `twotailt(x,df)`.

```
Cmd> twotailt(vector(-3.0795, 0.63278), nrows(x1)+nrows(y)-2)
(1)    0.0067965       0.5353
```

See Sec. 2.12.7 to see how to compute one-tail *P* values for these test statistics.

Functions `tint()` and `t2int()` return one- or two- sample confidence intervals based on Student's *t* computed from one or two REAL vectors and a specified confidence level or coverage. In a sense, they are complementary to `tval()` and `t2val()`. Their output is a REAL vector of length 2 containing the lower and upper ends of the confidence interval.

`tint(x,coverage)` returns a one-sample confidence interval for the population mean μ_x with confidence level `coverage` (a REAL scalar) based on data in the REAL vector `x`. For obvious reasons, 0 < `coverage` < 1. For instance, you can compute a 95% confidence interval by `tint(x,.95)` or `tint(x,1-.05)`.

`t2int(x,y,coverage)`computes a two-sample confidence interval for the difference $\mu_x - \mu_y$ of the population means of the data in REAL vectors `x` and `y`, where `coverage` is as for `tint()`. Like `t2val()`, function `t2int()` assumes equal variances and uses a pooled estimate of the common variance in computing the standard error of $\bar{x} - \bar{y}$. See below for usage that does not require equal variances.

```
Cmd> tint(x1,.95)
(1)       3.3521        5.9207  95% confidence interval for μ
Cmd> t2int(x1,y,.95)
(1)      -4.1936      -0.78364  95% confidence interval for μx-μy
```

Both `t2val()` and `t2int()` assume by default that $\sigma_x^2 = \sigma_y^2 = \sigma^2$, where σ_x^2 and σ_y^2 are the variances of the two populations being compared. Because of this assumption they use the estimated standard error of $\bar{x} - \bar{y}$, $s_{\bar{x}-\bar{y}} = s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$, where $s_p^2 = \frac{(n_1-1)s_x^2 + (n_2-1)s_y^2}{n_1 + n_2 - 2}$ is the "pooled" estimate of σ^2. When the variances cannot be assumed equal, you can use the "unpooled" standard error $s_{\bar{x}-\bar{y}} = \sqrt{\frac{s_x^2}{n_1} + \frac{s_y^2}{n_2}}$.

Unfortunately Student's *t*-distribution is no longer directly applicable. However, a good approximation is to use Student's *t* with estimated degrees of freedom $f = \frac{(v_x + v_y)^2}{\frac{v_x^2}{n_1 - 1} + \frac{v_y^2}{n_2 - 1}}$ where $v_x = s_x^2 / n_1$ and $v_y = s_y^2 / n_2$ (Snedecor and Cochran 1980, Sec. 6.11). `t2val(x,y,pooled:F)` computes the two sample t and `t2int(x,y,pooled:F)` the confidence interval using this approximation. The result of `t2val(x,y,pooled:F)` is a structure with components `t` and `df`.

```
Cmd> t2int(x1,y,.95,pooled:F) # slightly different from before
(1)      -4.1205     -0.85676

Cmd> t2val(x1,y,pooled:F) # not assuming same variances
component: t
(1)      -3.2186       Value of t-statistic
component: df
(1)       16.926       Estimated degrees of freedom
```

2.12.7 **P-values and cumulative distribution functions - cumxxx() functions** MacAnova has several functions for computing the cumulative distribution functions (CDFs) of standard probability distributions. You can use them to compute *P* values associated with many standard test statistics.

The first argument to these functions is always a value `x` of the random variable of interest. This may be followed by the value of parameters and sometimes keyword phrases as indicated in the following table:

Distribution	Usage
Standard normal	`cumnor(x)`
Student's *t* on `df` degrees of freedom	`cumstu(x,df)`
Non-central Student's *t* on `df` degrees of freedom, noncentrality `delta`	`cumstu(x,df,delta)`
χ^2 on `df` degrees of freedom	`cumchi(x,df)`
Non-central χ^2 on `df` degrees of freedom, noncentrality `noncen`	`cumchi (x,df,noncen)`
F on `df1` and `df2` degrees of freedom	`cumF(x,df1,df2)`
Non-central *F* on `df1` and `df2` degrees of freedom, noncentrality `noncen`	`cumF(x,df1,df2,noncen)`
Gamma with shape parameter `alpha`	`cumgamma(x,alpha)`
Beta with parameters `alpha` and `gamma`	`cumbeta(x,alpha,gamma)`
Non-central beta with parameters `alpha` and `gamma`, noncentrality `noncen`	`cumbeta(x,alpha,gamma,noncen)`

Distribution	Usage
Binomial with `n` trials and success probability `p`	`cumbin(x,n,p)`
Poisson with mean `lambda`	`cumpoi(x,lambda)`
Studentized range for `k` groups, `df` error degrees of freedom	`cumstudrng(x,k,df[,epsilon:eps])`
Dunnett's maximum *t* for `k` groups, `df` error degrees of freedom, group sizes determined by `groupsizes` (default all equal); probability is two sided without `onesided:T`.	`cumdunnett(x,k,df, [,groupsizes] [,onesided:T] [,epsilon:eps])`

The noncentral versions of `cumchi()`, `cumstu()`, `cumF()` and `cumbeta()` are useful in computing the powers of tests, that is, the rejection probability when the null hypothesis is not true.

Each of these CDF functions compute a *lower tail* probability, that is, $P(X \le x)$, where X is the random variable and x is a fixed number.

These functions are often used to compute *P* values based on the observed value x_{obs} of a test statistic X. The lower tail probability $P(X \le x_{obs})$ that each function computes can be directly interpreted as a *P* value for a one-tail test that rejects for values in the left tail of the test statistic. For *upper tail* tests such as χ^2 and *F* that reject the null hypothesis only for large positive values, you often need an upper tail *P* value $P(X \ge x_{obs}) = 1 - P(X < x_{obs})$. For example, `1 - cumchi(15.3,10)` computes the *P* value for a χ^2 statistic on 10 degrees of freedom with observed value 15.3. Of course, for the binomial and Poisson distributions which are discrete, when x_{obs} is an integer , $P(X \ge x_{obs}) = 1 - P(X \le x_{obs} - 1)$.

To compute a two-tail *P* value for a *z*- or *t*-test, use `2*(1-cumnor(abs(xobs)))` or `2*(1-cumstu(abs(xobs),df))`. You can compute a two-tail *P* value for an *F*-test of equal variance based on estimated variances `var1` and `var2` as

```
2*min(cumF(var1/var2,df1,df2), 1-cumF(var1/var2,df1,df2)).
```

All parameters except noncentrality parameters must be positive. Parameter `noncen` must be non-negative for `cumchi()`, `cumF()` and `cumbeta()`. Parameter `n` for `cumbin()` and `k` for `cumstudrng()` must be integers, but other parameters, including degrees of freedom, need not be integers. For `cumbin()`, `n` must be a positive integer and `p` must be between 0 and 1.

Parameter `groupsizes` for `cumdunnett()` is optional and differs from other parameters in that its value may be a vector of group sizes. If its length is less than `k`, the number of groups, its last non-zero element is replicated. Thus, when, say, `k` is 4, `vector(5,3)` or `vector(5,3,0,0)`is equivalent to `vector(5,3,3,3)`. If its length is longer than the number of groups, the extra elements are ignored, except that it is an error to have a non-zero element following a zero. Actually, only the relative sizes of its elements are used so `vector(5,3,2)` and `vector(.5,.3,.4)` are equivalent.

For all functions, either x or any of the parameters (but not keyword values) may be vectors, matrices or arrays. The sizes and shapes of the arguments must match, except that x or any of the parameters may always be a scalar, and the result is a variable of the same size and shape. If x or any of the parameters are scalars, they are treated as if they were the same size as other arguments with all elements the same. Thus, for example, `cumchi(5,run(3))` is equivalent to `cumchi(rep(5,3),run(3))`.

It is slightly different for `cumdunnett()` parameter `groupsizes`, which is treated as one or more vectors whose length is the last dimension of `groupsizes`. When `groupsizes` is a true vector (`ndims(groupsizes)` = 1) it is treated similarly to how scalar parameters are treated, being replicated to match the dimensions of the result; when `ndims(groupsizes)` > 1, `groupsizes` is treated as an "array" of dimension `ndims(groupsizes)-1` each of whose elements is a vector whose length is the last dimension of `groupsizes`. The first `ndims(groupsizes)-1` of `groupsizes` must match the dimensions of any other non-scalar argument. This is one of the few situations in which a vector of length *m* is not treated the same as a *m* by 1 matrix. The former is interpreted as specifying sizes for up to *m* groups while the latter specifies *m* scalar values for `groupsizes`, which would imply equal groups sizes, the default.

Keyword phrase `epsilon:eps` on `cumstudrng()` and `cumdunnett()` affects the accuracy of the result. Within limits, the smaller `eps` is, the more accurate the result, although the computation may take longer.

```
Cmd> vector(cumnor(1.96), cumstu(-2.1,10)) # normal, Student's t(10)
(1)        0.975     0.031039

Cmd> 2*(1 - cumstu(abs(-2.1),10)) # 2-tail t P-value
(1)     0.062077

Cmd> vector(1-cumchi(34,25),1-cumF(4.5,2,10)) # chi-squared and F
(1)      0.10791     0.040386

Cmd> 1 - cumstu(2.1,vector(10,20,30,40,50)) # 5 d.f.s at once
(1)     0.031039     0.024309     0.022121     0.021041     0.020398

Cmd> xbar <- vector(150.91,150.77,159.51,156.43,165.54)#enter means

Cmd> spooled <- 15.974 # enter pooled var from k=5 groups of n=20

Cmd> n <- 20; k <- 5

Cmd> studrng <- (max(xbar) - min(xbar))/(spooled/sqrt(n));studrng
(1)       4.1351   Studentized range statistic

Cmd> 1 - cumstudrng(studrng, k, k*(n-1)) # compute P value
(1)     0.034262   Significant at 5% level

Cmd> groupsizes <- matrix(vector(6 ,6, 6, 12, 3, 3),3)';groupsizes
(1,1)            6            6            6
(2,1)           12            3            3

Cmd> cumdunnett(2.5,3, 12, groupsizes)
(1)      0.94979      0.94678

Cmd> vector(cumdunnett(2.5,3,12), cumdunnett(2.5,3,12,vector(4,1)))
(1)      0.94979      0.94678
```

The last two commands have the same values because only the ratios of the elements of the group sizes parameter matter. Thus `12` is equivalent to `6` which is equivalent to row 1 of `groupsizes`, and `vector(4,1)` is equivalent to `vector(12,3)` which is equivalent to row 2 of `groupsizes`.

You can compute binomial and Poisson probabilities of the form P($X = x$) by taking differences of cumulative probabilities.

```
Cmd> p <- .3; N <- 9

Cmd> p8 <- cumbin(8,N,p) - cumbin(7,N,p); p8
(1)   0.00041334   P(X=8)

Cmd> (N/1)*p^8*(1-p)^(N-8) # (N/1) is binomial coefficient (N  1)
(1)   0.00041334   Confirmation of value
```

You can even compute P($X = 0$),..., P($X = n$) in a single line.

```
Cmd> cdf <- cumbin(run(0,N),N,p); cdf - vector(0,cdf[-(N+1)])
 (1)     0.040354      0.15565      0.26683      0.26683      0.17153
 (6)     0.073514     0.021004    0.0038579   0.00041334   1.9683e-05
```

The studentized range is R/s, where $R = x_{\max} - x_{\min}$ is the range from a normal sample of size `k` with standard deviation σ, and s^2 is an independent estimate of σ^2 with `df` degrees of freedom. It is often applied with data from k independent samples each of size n, with R/s, $= \{(\bar{x})_{\max} - (\bar{x})_{\min}\}/\{s_{\text{pooled}}/\sqrt{n}\}$, where s^2_{pooled} is the error mean square from an analysis of variance.

Dunnett's t is $\max_{2 \le j \le k} |t_{1,j}|$ (two-tail) or $\max_{2 \le j \le k} t_{1,j}$ (one-tail as specified by `onesided:T`), where $t_{1,j} = \dfrac{\bar{x}_1 - \bar{x}_j}{s\sqrt{\frac{1}{n_1} + \frac{1}{n_j}}}$. The $\bar{x}_j$ are sample means of independent random sample of size n_j from a N(0,σ^2) population, and s^2 is an independent estimate of σ^2 with `df` degrees of freedom. The optional argument `groupsizes` corresponds to $[n_1, n_2, \ldots, n_k]$. In most applications of Dunnett's t, group 1 is a control which is being compared with $k - 1$ treatments in groups 2 through k. Often $n_1 > n_2 = n_3 = \ldots = n_k$.

Non-central t is the ratio $\dfrac{z + \delta}{\sqrt{\frac{1}{f}\chi^2_f}}$, where z is standard normal, independent of χ^2_f in the denominator.

Non-central $\chi^2_f(\lambda)$ with non-centrality λ is $\sum_{i=1}^{f} (z_i + \delta_i)^2$, where $z_1, z_2, \ldots, z_f$ are independent standard normal and $\lambda = \sum_{i=1}^{f} \delta_i^2$.

Non-central $F_{f,g}(\lambda) = \dfrac{\chi^2_f(\lambda)/f}{\chi^2_g/g}$, where the numerator and denominator are independent.

Non-central $\beta_{\alpha,\gamma}(\lambda) = \dfrac{\chi^2_f(\lambda)}{\chi^2_f(\lambda) + \chi^2_g}$, $f = 2\alpha$, $g = 2\gamma$, where the two χ^2's are independent.

2.12.8 **Probability points and inverse cumulatives – invxxx()** functions MacAnova can compute probability points or critical values (inverse cumulatives) for many of the distributions in Sec. 2.12.7. Most require you to specify degrees of freedom or shape parameters. The first argument to these functions is always the probability value of interest. This may be followed by distribution parameters as indicated in the following table:

Distribution	Usage
Standard normal	`invnor(p)`
Student's *t* on `df` degrees of freedom	`invstu(p,df)`
χ^2 on `df` degrees of freedom	`invchi(p,df)`
Non-central χ^2 on `df` degrees of freedom and non-centrality parameter `lambda`	`invchi(p,df,lambda [,epsilon:eps])`
F on `df1` and `df2` degrees of freedom	`invF(p,df1,df2)`
Gamma with shape parameter `alpha`	`invgamma(p,alpha)`
Beta with parameters `alpha` and `gamma`	`invbeta(p,alpha,beta)`
Studentized range for `k` groups, `df` error degrees of freedom	`invstudrng(p,k,df [,epsilon:eps])`
Dunnett's maximum *t* for `k` groups, `df` error degrees of freedom, group sizes determined by `groupsizes` (default all equal); appropriate for two-tail test without `onesided:T`.	`invdunnett(p,k,df [,groupsizes] [.onesided:T,epsilon:eps])`

In every case, these inverse CDF functions compute a value x_0 such that $P(x \le x_0) = p$. That is, they are the inverses of the corresponding CDF functions in Sec. 2.12.7.

The value of `p` must always be between 0 and 1. All the other parameters except `k` for `invstudrng()`, including degrees of freedom, must be positive but need not be integers.

When `p` is a vector or matrix, the inverse CDF is computed for each element of `p` producing a vector or matrix. Similarly the parameters (but not keyword values) may be vectors or matrices. The sizes and shapes of the arguments must match, except that `p` or any of the parameters may always be scalars. Optional argument `groupsizes` for `invdunnett()` is an exception; see the discussion of `cumdunnett()` in Sec. 2.12.7.

Keyword phrase `epsilon:eps` on `invchi()`, `invstudrng()` and `invdunnett()` affects the accuracy of the result. Within limits, the smaller `eps` is, the more accurate the result, although the computation may take longer.

```
Cmd> invnor(vector(.10,.05,.025,.01,.005)) # normal prob points
(1)      -1.2816      -1.6449       -1.96      -2.3263      -2.5758

Cmd> invstu(.975,run(5,25,5)) # Student's t on 5, 10, 15, 20, 25 df
(1)       2.5706       2.2281       2.1314        2.086       2.0595

Cmd> invF(.95,5,run(5,25,5)) # F with df1=5, and df2=5,10,15,20,25
(1)       5.0503       3.3258       2.9013       2.7109        2.603

Cmd> invstudrng(1-vector(.1, .05, .01, .001), k, k*(n-1))
(1)       3.5309       3.9328       4.7373       5.7199

Cmd> invdunnett(vector(0.94979,0.94678),3,12,groupsizes)
(1)          2.5          2.5
```

The `invstudrng()` output consists of the 10%, 5%, 1% and 0.1% critical values of the Studentized range statistic based on 5 samples of size 20. The `invdunnett()` arguments match those used for `cumdunnett()` in Sec. 2.12.7. See Sec. 10.8.1 for another example.

You can actually compute a page of an *F*-table with a single command. The following computes the upper 5% point of *F* for numerator degrees of freedom 1 through 8 and denominator degrees of freedom 1 through 5 and then prints them out with rows and columns labeled by degrees of freedom.

```
Cmd> fvalues <- invF(1-.05,run(8)*rep(1,5)',rep(1,8)*run(5)')

Cmd> setlabels(fvalues,structure(enterchars(1 2 3 4 5 6 7 8),\
    enterchars(1 2 3 4 5))); fvalues
             1            2            3            4            5
1       161.45       18.513       10.128       7.7086       6.6079
2        199.5           19       9.5521       6.9443       5.7861
3       215.71       19.164       9.2766       6.5914       5.4095
4       224.58       19.247       9.1172       6.3882       5.1922
5       230.16       19.296       9.0135       6.2561       5.0503
6       233.99        19.33       8.9406       6.1631       4.9503
7       236.77       19.353       8.8867       6.0942       4.8759
8       238.88       19.371       8.8452        6.041       4.8183
```

See Sec. 8.4.1 for information on `setlabels()`.

2.12.9 **Grouping data in class intervals – bin()** An important way to summarize a data set is to compute a grouped frequency distribution. This consists of a table of the number of values that lie in each of a set of k class intervals defined by class boundaries $b_1 \le b_2 \le b_3 \le \ldots \le b_k \le b_{k+1}$. Class interval j consists of all values between b_j and b_{j+1}. Depending on who is making the definition, either the left end b_j or the right end b_{j+1} of the interval is considered in the interval. The counts in each interval are the *frequencies*. Function `bin()` allows you to make this summary. It is probably best explained by example.

```
Cmd> x <- vector(33,35, 41,41,42,44,47,49,49,50,50,50,\
51,53,53,54,54,54,55,57,58,60 ,61,62,67) # enter sorted data

Cmd> b <- vector(30,40,50,60,70)# or run(30,70,10); class limits

Cmd> bin(x,b) # group x using boundaries b
```

```
component: boundaries
(1)           30           40           50           60           70
component: counts
(1)            2           10           10            3
```

The first argument, x, contains the data to be grouped; the second, b, is a vector of class limits. The data don't need to be sorted, but have been here in order to make it easier to check that bin() is doing what it should. Vector b defines boundaries for 4 = length(b)-1 classes or groups. Group 1 consists of all values between b[1] and b[2], group 2 consists of all values between b[2] and b[3], and so on.

The result is a structure with components boundaries and counts. Component boundaries is identical with b, while counts[j] is the number of values of x values in class j, that is, the frequency of the class. Here there are two values (33 and 35) in class 1, ten values (41, 41, 42, 44, 47, 49, 49, 50, 50, 50) in class 2, and so on. The default convention is that any value exactly on a class limit at the *right* or upper end of a class is counted in that class. For example, the three 50's are considered to be in class 2 together with 41 through 49, and 60 goes in class 3.

Any value not in any class (here ≤30 or > 70) is not counted, but a warning message is printed. You can suppress the warning message by including silent:T as an argument.

```
Cmd> bin(vector(x,25,75),b) # data vector enlarged by 2 values
WARNING: 1 low and 1 high values not counted by bin()
component: boundaries
(1)           30           40           50           60           70
component: counts
(1)            2           10           10            3
```

Some statistics books recommend that you include the *left* or lower end in a class interval instead of the right end. You can follow this convention by using leftendin:T as an argument.

```
Cmd> bin(x,b,leftendin:T) # lower end of interval in class
component: boundaries
(1)           30           40           50           60           70
component: counts
(1)            2            7           12            4
```

Now the new frequencies reflect the fact that the three 50's are put in class 3 and the 60 in class 4.

When you want equally spaced boundaries, the usage, bin(x,vector(b0,width)), can simplify things. b0 is interpreted as a "typical" class limit and width is the desired class width. Enough boundaries of the form b ± *j*×width are computed so that *all* the data are included. b0 need not be in the range of the data.

```
Cmd> bin(x,vector(30,10)) # gives the same result as bin(x,b)
component: boundaries
(1)           30           40           50           60           70
component: counts
(1)            2           10           10            3
```

Instead of vector(30,10), you could use vector(0,10) or even vector(-100,10)

and still get the same output.

Still easier and OK for quick summaries is `bin(x,nclasses)`, where `nclasses` is the number of classes wanted. `bin()` chooses `nclasses`+1 equally spaced class limits which include all the data but they won't be "neat."

```
Cmd> bin(x,4) # bin() chooses boundaries
component: boundaries
(1)        32.15        41.5        50.85        60.2        69.55
component: counts
(1)            4           8           10           3
```

Even easier is simply `bin(x)` for which `bin()` selects both the number of classes and the width.

```
Cmd> bin(x)
component: boundaries
(1)        32.15       38.383       44.617        50.85       57.083
(6)       63.317        69.55
component: counts
(1)            2           4           6           8           4
(6)            1
```

You can use keyword phrase `leftendin:T` with all these usages.

If `x` is a `REAL` matrix, rather than a vector, each column is grouped separately, all with the same class limits. Component `boundaries` is as before, but now component `counts` is a matrix, with each column containing the frequencies for the data in the corresponding column of `x`.

See also `tabs()` (Sec. 3.12) which summarizes data cross-classified by one or more positive integer valued variables.

2.13 **Random numbers – runi(), rnorm(), rbin() and rpoi()** MacAnova has functions for generating standard normal, uniform, binomial and Poisson pseudo-random numbers. For each of them, the first argument, `N`, must be a positive integer.

`runi(N)` returns a vector of `n` pseudo-random variables that are *uniformly* distributed between 0 and 1. When `a` and `b` are scalars, `a+(b-a)*runi(N)` are uniform between `a` and `b`.

`rnorm(N)` returns a vector of `N` pseudo-random variables which have *standard normal* ($\mu = 0, \sigma = 1$) distribution. When `mu` and `sigma` are scalars, `mu+sigma*rnorm(N)` is a vector of `N` normal random variables with mean `mu` and standard deviation `sigma`.

`rbin(N,n,p)` returns a vector of `N` independent *binomial* pseudo-random variables with sample size n and probability of "success" `p`. `n` must be a positive integer scalar or a `REAL` vector of `N` positive integers. `p` must be a `REAL` scalar between 0 and 1 or a `REAL` vector of `N` values between 0 and 1. If `n` or `p` is a scalar, it is used for every element of the result. Otherwise, `n[i]` and/or `p[i]` are used for element `i` of the result.

`rpoi(n,lambda)` generates a vector of `n` independent pseudo-random variables with a *Poisson* distribution with mean `lambda`. `lambda` must either be a `REAL` scalar ≥ 0 or a `REAL` vector of length `N` with each `lambda[i]` ≥ 0. A scalar lambda is used for every

element of the result. Otherwise, element `i` of the result will be Poisson with mean `lambda[i]`.

See Sec. 2.13.1 for examples of these functions.

2.13.1 **Random number "seeding" – setseeds() and getseeds()** The values computed by `runi()`, `rnorm()`, `rbin()` and `rpoi()` depend on two positive integer "seeds" which are updated every time a number is computed. These are internal to MacAnova but can be set by either `setseeds()` or `setoptions()` (Sec. 8.1.2). If the seeds are the same on two different occasions you will get exactly the same sequence of random numbers.

You can set the seeds by `setseeds(vector(seed1,seed2))` or `setseeds(seed1, seed2)`, where `seed1` and `seed2` are positive integers between 1 and 2147483399. `setseeds(0,0)` initializes the seeds with values based on the date and time. This provides a more or less random starting point for random numbers. If you don't explicitly set the seeds, the first time random numbers are computed the seeds are initialized as if you had typed `setseeds(0,0)`.

You can retrieve current values of the seeds in a vector of length 2 by function `getseeds()`. If `getseeds()` is `vector(0,0)`, the seeds have not been set. Resetting the seeds to values previously retrieved makes it possible to generate the same sequence of random numbers more than once.

Here are some simple examples.

```
Cmd> runi(5) # 5 uniforms without initializing
WARNING: starting random number seeds are 1979189978 and 1730035780
(1)       0.2765     0.046009      0.27089      0.21016      0.16606

Cmd> rnorm(5) # 5 normals
(1)       1.9504     -0.23464     -0.58285      0.43007     -0.09322

Cmd> rpoi(5,run(5,25,5)) # 5 Poisson with means 5, 10, 15, 20, 25
(1)            6           16           10           16           23

Cmd> rbin(5,50,run(5,25,5)/50) # 5 Binomial, n = 50, same means
(1)            5            7           15           13           25

Cmd> getseeds() # value is "invisible"; output is "side effect"
Seeds are 1255236055 and 150892041

Cmd> setseeds(1979189978,1730035780) #reset earlier seeds,

Cmd> runi(5) #same values as before
(1)       0.2765     0.046009      0.27089      0.21016      0.16606
```

2.13.2 **Generating other random variables** Suppose X is a continuous random variable with cumulative distribution function $G(x) = P(X \le x)$ with inverse $H(p) = G^{-1}(p)$, that is $H(G(x)) = x$. Then, when U is a random variable uniformly distributed between 0 and 1, the random variable $Y = H(U)$ has the same distribution as X. You can use this fact to generate random samples for any continuous random variable for which you know how to compute the $H = G^{-1}$. But that is exactly what the functions such as `invstu()`, `invchi()` and `invF()`, described in Sec. 2.12.8, do. By using `runi(n)` as argument `p` in these inverse CDFs, you can generate pseudo-random samples of Student's t, F, χ^2, gamma, and beta distributed random variables, among

others.

```
Cmd> invF(runi(5),10,30); # small random sample from F(10,30)
(1)      0.72673       1.6098       2.4452      0.46853      0.76309
Cmd> invchi(runi(5),10,20) # noncentral chi-squared sample, 10 df
(1)       27.926       39.296       21.936         47.4       35.721
```

A number of distributions of counts can be represented as Poisson random variables with *random* mean λ. For example, if λ is a gamma distributed random variable with mean μ and shape parameter α, and, conditional on λ, X is Poisson with mean λ, then unconditionally, X has a negative binomial distribution, with mean μ and index α, that is

$$P(X = x) = (1-\mu/\alpha)^{\alpha}\frac{\alpha(\alpha+1)...(\alpha+x-1)}{x!}\left(\frac{\mu}{\alpha}\right)^{x}$$

Here is how you might use this fact to generate a small random negative binomial sample with mean 20 and index 4.

```
Cmd> rpoi(5,(20/4)*invgamma(runi(5),4))
(1)             5             9             8             9            19
```

Other discrete distributions can be represented as binomial random variables with random success probability p. If p has a beta distribution with density $\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}p^{\alpha-1}(1-p)^{\beta-1}$, $0 \le p \le 1$, the resulting distribution is known as the beta-binomial distribution which has probabilities

$$P(X = x) = \frac{n!}{x!(n-x)!}\frac{(\alpha)_x(\beta)_{n-x}}{(\alpha+\beta)_n},\ x = 0, 1, ..., n.$$

Here, using `rbin()` (Sec. 2.13) and `invbeta()` (Sec. 2.12.8), we generate a small beta-binomial random sample with n = 30, α = 3 and β = 10.

```
Cmd> rbin(5,30,invbeta(runi(5),3, 10))
(1)             4             7             3             5             0
```

2.14 **More on rep() and run()** Functions `run()` and `rep()` are particularly useful for setting up factor variables in ANOVA models (described in Chapter 3). As described in Sec. 2.8.12, `run()` generates equally spaced sequences and `rep()` replicates its first argument. We saw there that `run(n)` returned `vector(1,2,...,n)` and `rep(v,n)` returned `vector(v,v,...,v)`, where there are `n` repetitions and where `v` is a scalar or vector.

```
Cmd> rep(run(4),3) # 3 replicates of 1,2,3,4
 (1)             1             2             3             4             1
 (6)             2             3             4             1             2
(11)             3             4
```

There is a more general usage of `rep()`. Suppose `counts` is a vector of non-negative integers *of the same length* as `v`. Then, in the output returned by `rep(v,counts)`, the each `v[i]` is repeated `counts[i]` times. In particular, `rep(v,rep(m,n))`, where `n` is the length of `v`, produces `m` copies of each element of `v`.

```
Cmd> rep(run(3),rep(4,3)) # 4 copies of each element of run(3)
 (1)           1           1           1           1           2
 (6)           2           2           2           3           3
(11)           3           3
```

Thus you might use `rep(run(4),3)` and `rep(run(3),rep(4,3))` to generate treatment number and block numbers associated with a randomized block design with 4 treatments and 3 replicates, when all the data for each block is together, in the order of treatment number..

```
Cmd> rep(run(4),vector(2,1,0,6)) # 2 1's, 1 2, no 3's and 6 4's.
(1)           1           1           2           4           4
(6)           4           4           4           4
```

2.15 **Making graphs** MacAnova has several functions and pre-defined macros for making graphs of data. The basic functions are summarized in the following table:

Function or Macro	Description
`plot(x,y)`	Scatter plot or impulse plot of `y` versus `x`, points optionally connected by lines
`lineplot(x,y)`	Connected line plot of x versus y, optionally augmented by "impulses"
`chplot(x,y,sym)`	Scatter plot or impulse plot of `y` versus `x`, using user supplied symbols, points optionally connected by lines
`showplot()`	Redisplay previously plotted graph, possibly with changed labels or plotting limits
`colplot(x)`	Makes parallel line plots of matrix `x` against row number
`rowplot(x)`	Makes parallel line plots of matrix `x` against column number

"Impulses" are vertical lines connecting points with the *x*-axis.

`plot()`, `lineplot()` and `chplot()` take at least two arguments `x` and `y`, where `x` is a REAL vector and `y` is a REAL vector or matrix; they plot each column of `y` against the values of `x`. The length of `x` ordinarily must match the number of rows in `y`, but see below for a useful exception.

Instead of arguments `x` and `y`, you can use a structure with at least two components, the first two of which are interpreted as `x` and `y`. Any extra components are ignored. For example, `plot(x,y)` and `plot(structure(x,y))` are equivalent.

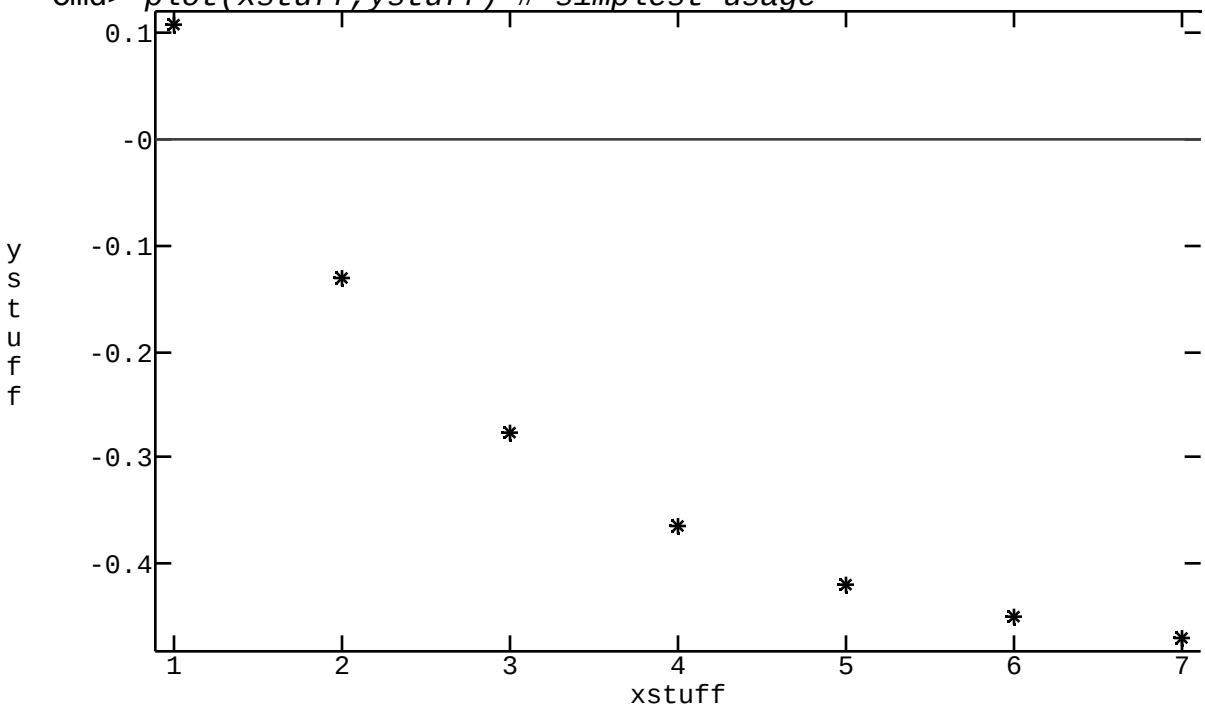
All plotting commands also recognize several optional keyword arguments which you can use to specify a title, axis labels, and minimum and maximum values to be plotted. See Sec. 2.15.5 and 8.5.

In windowed versions of MacAnova (Macintosh, Windows, Motif) high resolution graphs are drawn in separate windows. In the Macintosh and Windows versions, you can copy a graph window to the Clipboard and then paste it into a word processor or graph editor document. You can print graph windows using **Print Graph** on the **File**

menu (see Sec. B.3.2, D.3.1 and F.3.1 B in the Appendices).

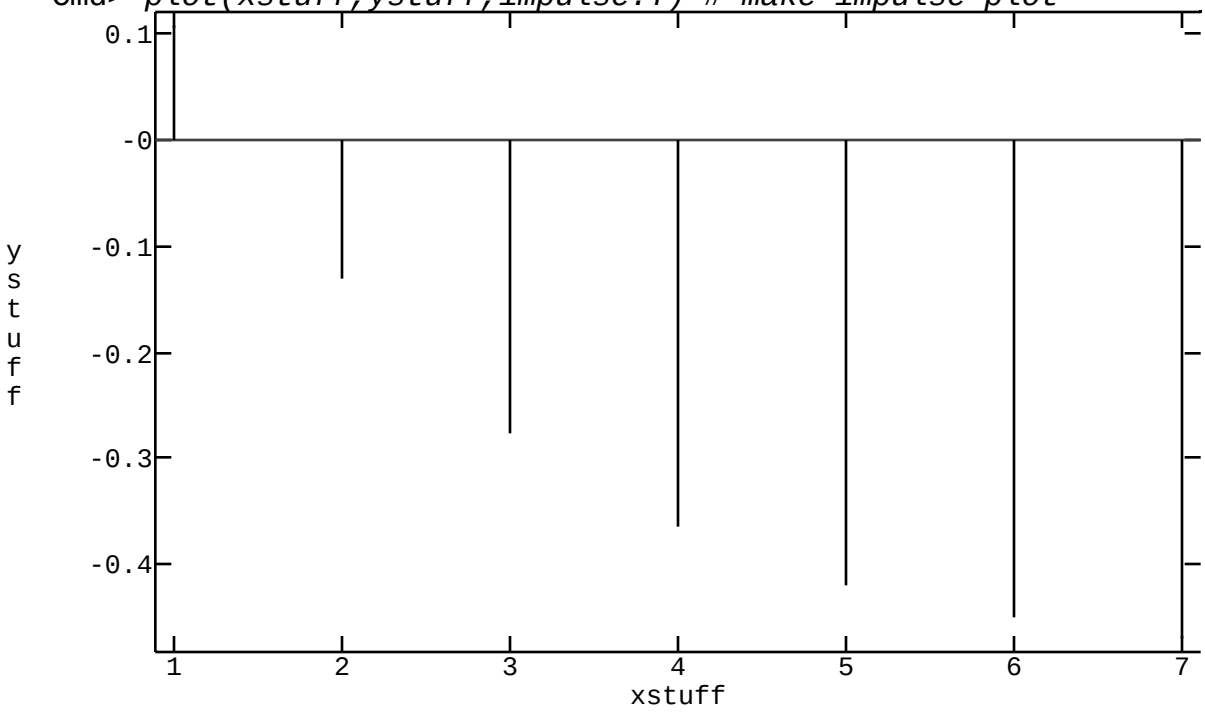
2.15.1 **plot()** `plot(x,y)` does a simple scatter plot of `y` versus `x`. If `y` has several columns, a different plotting symbol is used for each column.

```
Cmd> xstuff <- vector(1,2,3,4,5,6,7)
Cmd> ystuff <- exp(-.5*xstuff) - .5
Cmd> plot(xstuff,ystuff) # simplest usage
```



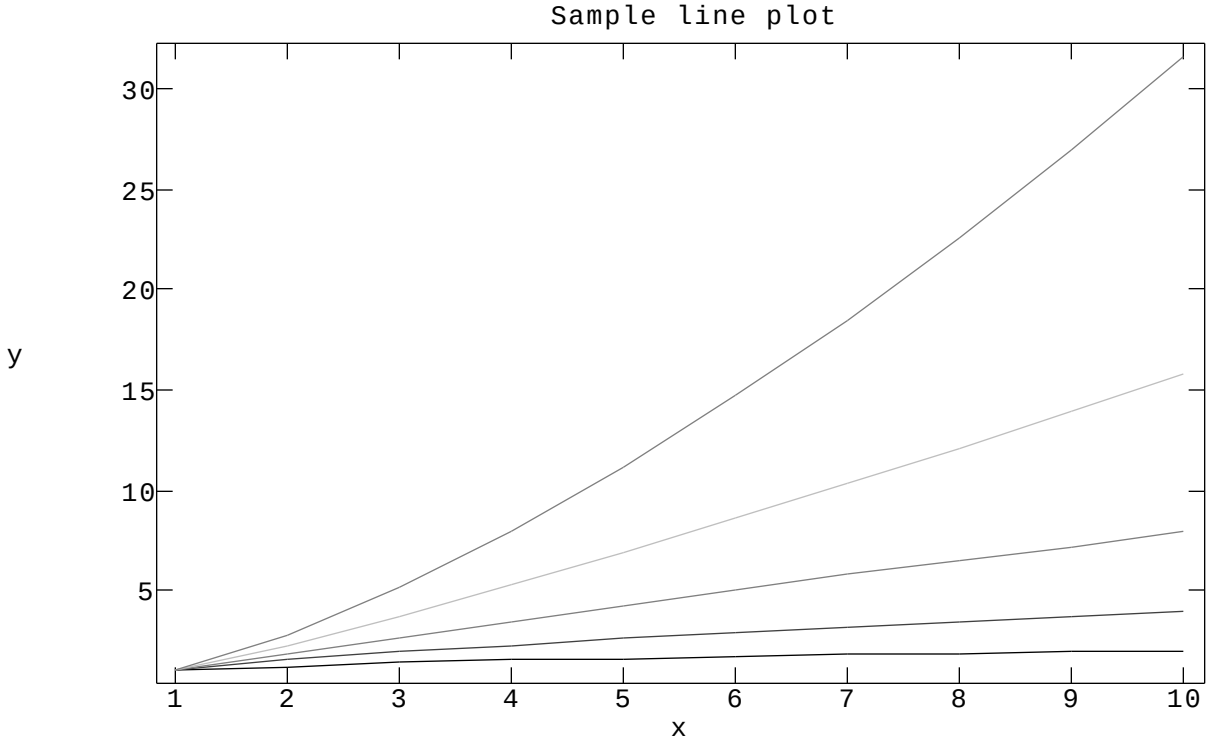
Note that the axes are labelled with the variable names.

```
Cmd> plot(xstuff,ystuff,impulse:T) # make impulse plot
```



2.15.2 **lineplot()** `lineplot(x,y)` draws lines between the successive points plotted, but does not draw any symbol at the points. Ordinarily, the elements of `x` should be in monotonic (increasing or decreasing) order. `lineplot()` uses different line types (solid, dashed, dotted, ...) for successive columns of `y`. You can use keyword `linetype` to change the default line types. See Sec. 8.5.1.

```
Cmd> lineplot(x:run(10),y:run(10)^(.3*run(5)'),\
title:"Sample line plot") #Note keywords x and y to label axes
```



2.15.3 **chplot()** `chplot(x,y,sym)` is similar to `plot(x,y)`, except that REAL or CHARACTER vector or matrix `sym` specifies plotting symbols or characters. When `sym` is a matrix, it must have the same number of columns as `y`.

When `sym` is REAL, its elements must be integers between 1 and 999 and the numbers themselves are used as the plotting symbols.

When `sym` is a CHARACTER vector or matrix, its elements are used as plotting symbols (actually only the first three characters are used if there are more). For example, `chplot(run(5),run(5)^2, vector("Cat","Dog","Wolf","Sheep","Bird"))` labels the successive points with `Cat`, `Dog`, `Wol`, `She` and `Bir`.

Character symbols `"\1"`, `"\2"`, `"\3"`, `"\4"`, `"\5"`, `"\6"`, `"\7"`, `"\10"` are special and are interpreted as symbols ◇, +, ⊡, ×, △, ✳, ·, and ⋆. For example, when `y` is a vector, `chplot(x,y, "\6")` is identical to `plot(x,y)` and `chplot(x,y, "\7")` places a dot at each point.

When `sym` has a single element as in `chplot(x,y,"#")`, it is used for all points.

When `sym` is a vector whose length matches the number of columns in `y`, then `sym[j]` labels all points associated with *column* `j`, that is the point `y[i,j]` is plotted against `x[i]` with character or number `sym[j]`. For example, if `y` has 3 columns, `chplot(x,y,run(3))`, labels all the points from `y[,1]`, `y[,2]` and `y[,3]` with `"1"`, `"2"` and `"3"`, respectively.

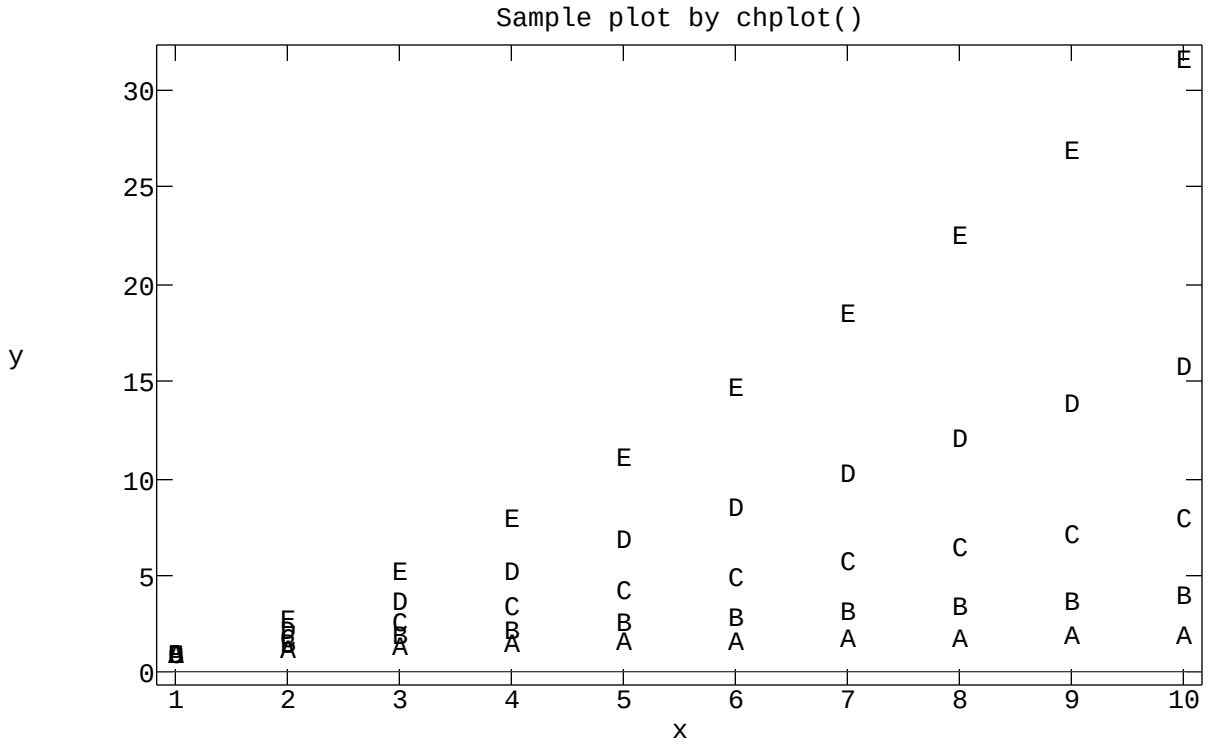
When `sym` is a vector whose length does *not* match the number of columns in `y`, then `sym[j]` labels *row* `j`, with the symbols *repeating cyclically* if `length(sym)` <

`nrows(y)`.

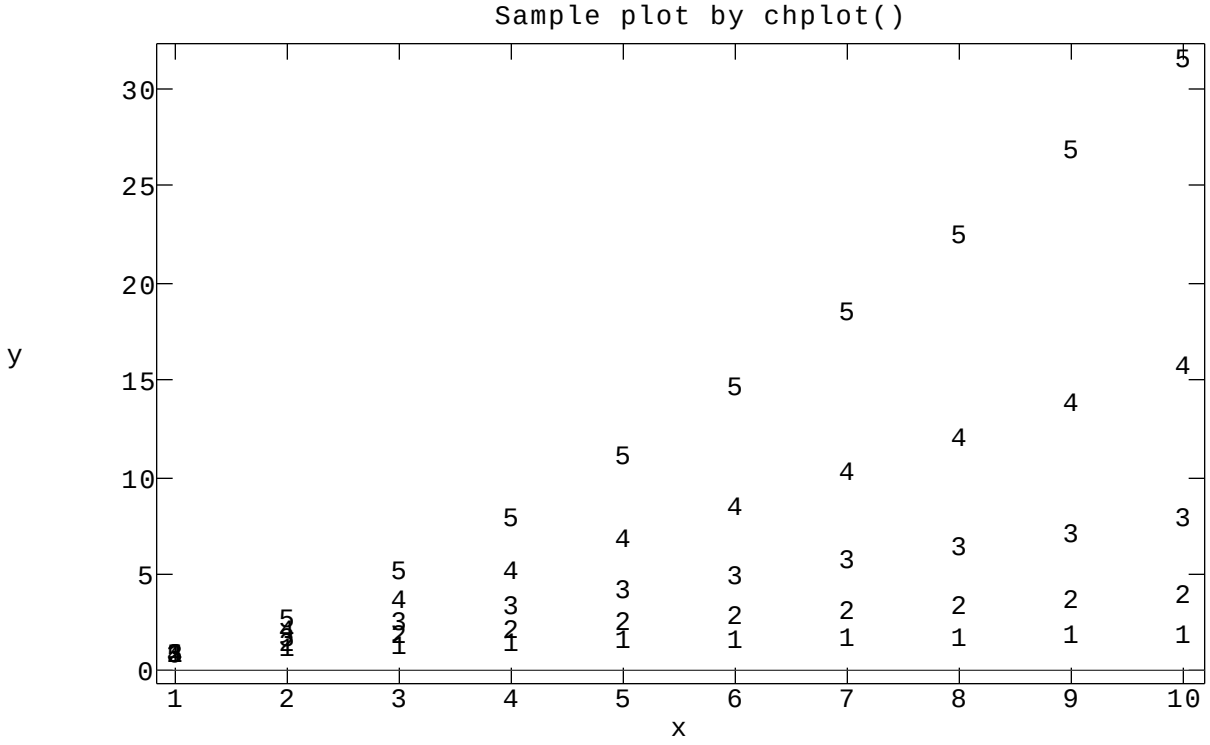
Finally, when `sym` is a matrix it must have the same number of columns as `y`, and symbols for plotting the values from row *i* of `y` are taken from row *i* of `sym`, repeating rows of `sym` cyclically if necessary.

You can omit `sym` altogether; when `y` has only one column, each point is labelled with the row number and when `y` has several columns, all the points from a column are labelled with the column number.

```
Cmd> chplot(x:run(10),y:run(10)^(.3*run(5)'),\
vector("A","B","C","D","E"), ymin:0,\
title:"Sample plot by chplot()")
```



```
Cmd> chplot(x:run(10),y:run(10)^(.3*run(5)'),ymin:0,\
title:"Sample plot by chplot()") # no plotting symbols provided
```



2.15.4 **Equally spaced x values** The exception to the requirement that `x` and `y` have the same number of rows is a convention you can use when plotting the rows of `y` against *equally spaced* values on the *X*-axis. When `x` is a `REAL` *scalar* (single number), `y[i,]` is plotted against `vector(x,x+1,x+2,...)`, that is they are equally spaced by 1 starting with `x`. When `x` is a *vector of length 2*, say, `vector(x0,d)`, `y[i,]` is plotted against vector(`x0,x0+d, x0+2*d,...`), that is equally spaced starting with `x0` and incrementing by `d`. For example, `plot(1,y)` is equivalent to `plot(run(nrows(y)),y)` and `lineplot(vector(1967,1/12),y)` might be used to plot monthly values for which `y[1,]` was data for January, 1967.

2.15.5 **Graphics keywords** See Sec. 8.5 for information on the use of keywords `xmin`, `ymin`, `xmax`, `ymax`, `xlab`, `ylab`, `xaxis`, `yaxis`, `title`, `file`, `keep`, and `show`, and on how to add information to previously created graphs.

You can draw lines between points plotted by `plot()` or `chplot()` by using keyword phrase `lines:T`. If you use `lines:F` on `lineplot()`, it becomes equivalent to `plot()`.

On all three commands, you can use `impulses:T` add "impulses" to the plot. On `plot()` but not on `chplot()`, this suppresses the plotting symbols.

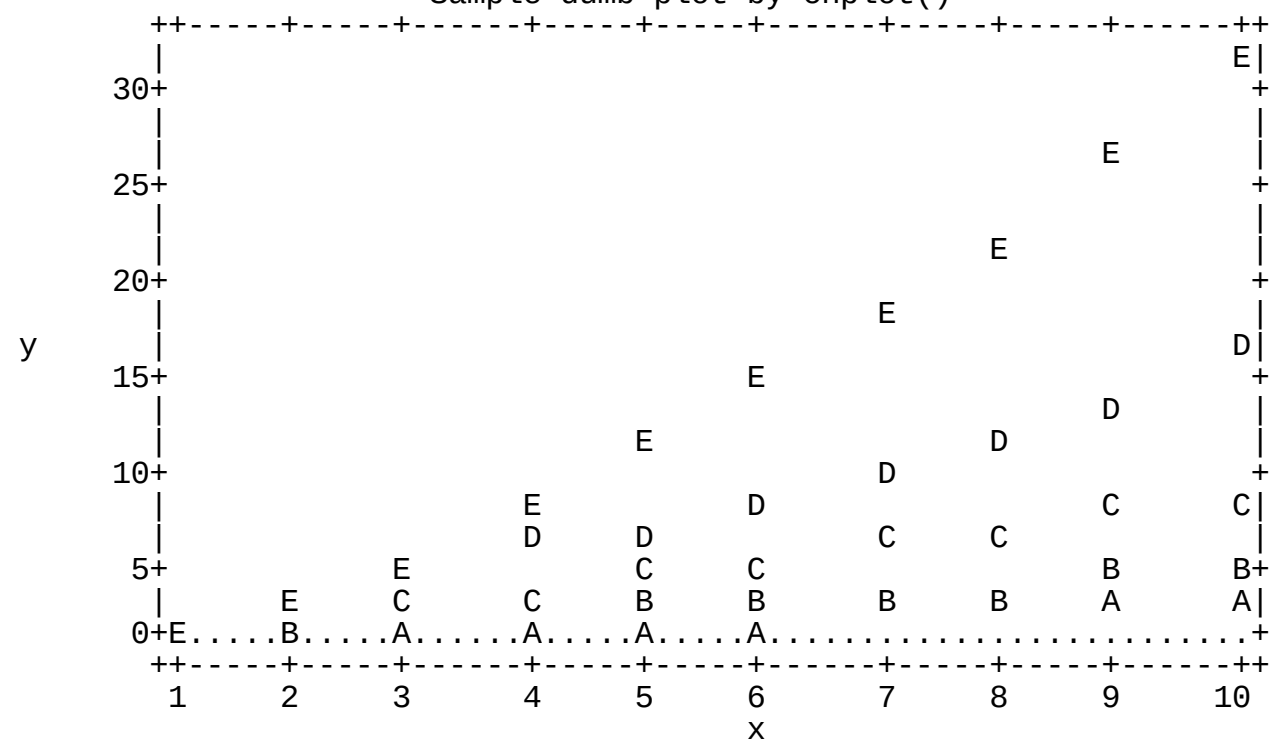
If you use `lines:F` on `lineplot()`, it becomes the same as `plot()`.

2.15.6 **colplot and rowplot** These are pre-defined macros that use `chplot()` to draw "interaction plots" of data in a matrix `x`. `colplot(x)` draws line plots of each *column* of `x` versus the row numbers `run(nrows(x))`. Data points are labelled with the

column number. Similarly `rowplot(x)` draws line plots of each *row* of `x` versus the column numbers `run(ncols(x))`. See Sec. 10.12 for an example of the use of `colplot`.

2.15.7 **Low resolution plots** You can use keyword phrase `dumb:T` as an argument with any plotting command, as in `plot(x,y,dumb:T)`. This results a low resolution plot made up of letters and other printable symbols. For many purposes this is sufficient to get a clear picture of the relationships being plotted, and has the advantage that it is printable on any printer and is written along with other output to a spool file (see. Sec. 2.16).

```
Cmd> chplot(x:run(10),y:run(10)^(.3*run(5)'),\
vector("A","B","C","D","E"),ymin:0,\
title:"Sample dumb plot by chplot()",dumb:T)
```



You can use keywords `height` and `width` to override the default plot size. For example, a graph produced by `plot(x,y,dumb:T,height:15,width:50)` consists of 15 lines of up to 50 characters, including the title and labels.

2.16 **Using spool() to save output** You often want to keep track of exactly what you have done and what were the results. Command `spool()` allows you to "spool" your session, that is, save in a file a transcript of what you typed and what MacAnova printed. Spooling is a little bit like using a tape recorder – everything, wanted or unwanted is recorded. After leaving MacAnova, you can can print the spool file, or more usually, incorporate it in a report using a word processor.

`spool(fileName)`, where `fileName` is a quoted string or `CHARACTER` variable containing a legal file name, starts spooling on the named file. From that point on until you quit MacAnova or suspend spooling, everything you type and everything MacAnova responds (except high resolution graphs) is written to the file in plain text

(ASCII) form. In a windowed version (Macintosh, Windows or Motif), if `fileName` is the empty name `""`, you will be able to specify the file using a dialog box. On a Macintosh, selecting **Spool Output To File** on the **File** menu is another way to start spooling.

If the spool file already exists, `spool()` normally starts writing at its end. This allows you to accumulate output from several runs in a single file. If this is not what you want, use `spool(fileName,new:T)` which starts transcription at the start of the file, destroying any information already there.

To suspend spooling, simply type `spool()`, with no argument. On a Macintosh you can select **Stop Spooling** on the **File** menu. If spooling has been suspended, `spool()`, with no argument restarts it. On a Macintosh you can select **Resume Spooling** on the **File** menu.

For example, you might suspend spooling while you experiment with different analyses of data, and then resume it once you have a better idea of what you plan to do.

```
Cmd> spool("spool.txt") # start spooling on file 'spool.txt'

Cmd> getseeds() #display current seeds, see Sec. 2.13
Seeds are 946682807 and 873681665

Cmd> x <- runi(5); x # this will be transcribed on spool.txt
(1)      0.38959      0.71524      0.63019      0.55814       0.8608

Cmd> spool() # suspend spooling
Spooling on spool.txt suspended

Cmd> x-3 # this will not be transcribed on spool.txt
(1)      -2.6104      -2.2848      -2.3698      -2.4419      -2.1392

Cmd> spool() # resume spooling.  This line will not be transcribed
Resume spooling on spool.txt

Cmd> x+1 # this line and following output will be transcribed
(1)       1.3896       1.7152       1.6302       1.5581       1.8608
```

This is what ends up in file `spool.txt`.

```
Cmd> getseeds() #display current seeds, see Sec. 2.13
Seeds are 946682807 and 873681665

Cmd> x <- runi(5); x # this will be transcribed on spool.txt
(1)      0.38959      0.71524      0.63019      0.55814       0.8608

Cmd> spool() # suspend spooling
Spooling on spool.txt suspended

Cmd> x+1 # this line and following output will be transcribed
(1)       1.3896       1.7152       1.6302       1.5581       1.8608
```

In the windowed versions of MacAnova, it is also possible directly to save the contents of the command window to a file using **Save Window** or **Save Window As...** on the **File** menu. See Appendices B, D and F.

2.17 **Using save() and restore() to preserve work between sessions** All the variables and macros you are currently using are referred to collectively as the *workspace*. During a MacAnova run, the workspace is in computer memory (RAM) and not in a file on disk. When you quit MacAnova, unless you take precautions, your workspace and possibly the result of a lot of work is normally lost. You can preserve everything in your workspace on disk by `save(fileName)`, where `fileName` is a quoted string or `CHARACTER` variable specifying the name of the file. On a later MacAnova session, `restore(fileName)` restores the workspace to the way it was when saved. As usual in windowed versions, the empty file name "" lets you select a file.

```
Cmd> x # show that vector x is defined
(1)      0.67374      0.13423      0.82378      0.89615      0.66544

Cmd> save("session1.sav") # save workspace, including vector x
Workspace saved on file session1.sav

Cmd> quit  # terminate the run
```

Now suppose you start a new MacAnova session.

```
Cmd> x # At this point x is not defined
UNDEFINED

Cmd> restore("session1.sav") # restore the save file
Restoring workspace from file session1.sav
Workspace saved Thu Oct 30 11:52:58 1997

Cmd> x # variable x as we previously saved it is now defined
(1)      0.67374      0.13423      0.82378      0.89615      0.66544
```

The file created by `save()` is in a binary format readable only by MacAnova running on the same type of computer and is useless on any other type of computer. In contrast, `asciisave(fileName)` saves your workspace in a form that can be read by MacAnova on any type of computer. The file it writes is a plain text or ASCII file although it has an arcane format intended to be read only by a computer. You may even be able to send it via e-mail with little danger of corruption since there are no non-printable characters in the file. Command `restore()` can read either type of file although a file written by `asciisave()` may take a little longer to restore. The file created by `asciisave()` is often larger than the one created by `save()`, although that need not be the case.

Another use for `save()` and `restore()` is when you are running MacAnova on an unstable computer, subject to crashes, or running a Unix version over the telephone using a modem where there is a chance the connection will be broken. If you use `save()` every few minutes, you minimize the danger of losing work. To make this easier, after the first use of `save(fileName)`, simply `save()`, omitting `fileName`, saves to the same file. If the original saving was done by `asciisave()`, a plain text files will be written.

To save your workspace on a windowed version, select **Save Workspace** on the **File** menu or press **⌘K** on a Macintosh or Ctrl+K in the Windows or Motif versions. The first time you do this, you will be prompted for a save file. Later uses of **Save Workspace** will save on the same file with no prompting.

See Sec. 7.7 for information on partial saves and keywords `all`, `delete`, options and `history`.